Pre-Proceedings

15th International Workshop on Logical and Semantic Frameworks, with Applications

LSFA 2020

August 27-28, 2020

Editors:

Cláudia Nalon & Giselle Reis

Contents

Preface v

SANDRO PRETO AND MARCELO FINGER
An Efficient Algorithm for Representing Piecewise Linear Functions into Logic ... 1

CARLOS OLARTE, ELAINE PIMENTEL AND BRUNO XAVIER
A fresh view of linear logic as a logical framework ... 16

ERICK GRILO AND BRUNO LOPES (Short Paper)
ReLo: a dynamic logic to reason about Reo circuits ... 32

ALEXIS MARTÍN, ALEJANDRO RIOS AND ANDRÉS EZEQUIEL VISO
Pure Pattern Calculus a la de Bruijn ... 39

SANDRA ALVES, MARIBEL FERNANDEZ AND MIGUEL RAMOS
EVL: a typed higher-order functional language for events ... 52

SUSANNAH MANSKY AND ELSA GUNTER
Safety of a Smart Classes-Used Regression Test Selection Algorithm .. 68

SEPEHR AMIR-MOHAMMADIAN AND CHADI KARI
Correct Audit Logging in Concurrent Systems ... 82

LINDA BRODO AND CARLOS OLARTE
A constraint-based language for multiparty interactions ... 99

SANDRA ALVES, DANIELE NANTES-SOBRINHO, DANIEL VENTURA AND JORGE A. PÉREZ (Short Paper)
An Investigation of Linear Substitution λ-Calculus as Session-Typed Processes ... 115

GUILHERME GOMES FELIX DA SILVA, EDWARD HERMANN HAEUSLER AND BRUNO LOPES (Short Paper)
Formalization of Cryptographic Algorithms in the Lean Theorem Prover 122

ALBERTO PARDO AND CECILIA MANZINO
Agda formalization of a security-preserving translation from flow-sensitive to flow-insensitive security types ... 128

SEBASTIÁN URCIUOLI, ALVARO TASISTRO AND NORA SZASZ
Strong Normalization for the Simply-Typed Lambda Calculus in Constructive Type Theory Using Agda ... 142

Preface

This document contains selected papers from the **15th International Workshop on Logical and Semantic Frameworks, with Applications** (LSFA 2020). The workshop was planned to take place in Salvador, Brazil. Due to the global pandemic, the two-day event was held entirely online on August 27 and 28, 2020. Besides the full regular papers, this volume contains accepted submissions in the form of short papers, such as system descriptions, proof pearls, rough diamonds (preliminary results and work in progress), original surveys, or overviews of research projects, where the focus is more on elegance and dissemination than on novelty. Regular papers are published in ENTCS. The programme committee has accepted nine regular papers out of 13 submissions (17 abstracts); and three short papers from four submissions (six abstracts). All papers were reviewed by at least three members of the programme committee.

Logical and semantic frameworks are formal languages used to represent logics, languages and systems. These frameworks provide foundations for the formal specification of systems and programming languages, supporting tool development and reasoning. The objective of LSFA, a series of annual events starting in 2006, is to gather together theoreticians and practitioners to promote new techniques and results from the theoretical side, and feedback on the implementation and use of such techniques and results from the practical side. Topics of interest to LSFA include, but are not limited to: specification languages and meta-languages, formal semantics of languages and logical systems, logical frameworks, semantic frameworks, type theory, proof theory, automated deduction, implementation of logical or semantic frameworks, applications of logical or semantic frameworks, computational and logical properties of semantic frameworks, logical aspects of computational complexity, lambda and combinatory calculi, and process calculi.

We are grateful for the efforts of the *programme committee* who provided detailed reviews and fruitful discussions during the double reviewing phase of the papers, which assured the high technical quality of the event. In 2020, the programme committee included:

- Beniamino Accattoli (INRIA Saclay);
- Sandra Alves (University of Porto);
- Mario Benevides (UFF);
- Ana Bove (Chalmers);
- Manuela Busaniche (CONICET-UNL);
- Marco Cerami (UFBA);
- Amy Felty (University of Ottawa);
- Maribel Fernández (King's College London);

- Francicleber Ferreira (UFC);
- Renata de Freitas (UFF);
- Lourdes del Carmen González Huesca (UNAM);
- Edward Hermann Haeusler (PUC-Rio);
- Oleg Kiselyov (Tohoku University);
- João Marcos (UFRN);
- Alberto Momigliano (University of Milano);
- Daniele Nantes-Sobrinho (UnB);
- Carlos Olarte (UFRN);
- Revantha Ramanayake (TU Wien);
- Camilo Rocha (PUJ);
- Nora Szasz (Universidad ORT);
- Ivan Varzinczak (Université d'Artois);
- Daniel Ventura (UFG);
- Renata Wassermann (USP).

We are also grateful for the *subreviewers* who helped in the reviewing process: Ernesto Copello (Universidad ORT), Isaque Lima (UFRJ), Jorge A. Pérez (University of Groningen), Álvaro Tasistro (Universidad ORT), and Nachiappan Valliappan (Chalmers).

We are in debt with the local organisation for their support: Aline Andrade (UFBA, Meeting Chair), Andreas Brunner (UFBA), Adolfo Duran (UFBA), Samuel Gomes (UFBA), Bruno Lopes (UFF), Darllan Pinto (UFBA), Lais Salvador (UFBA), and Marlo Souza (UFBA). We also acknowledge and thank the scientific support by the Brazilian Logic Society and the financial support by 2BI Tecnologia and VOX Tecnologia da Informação LTDA.

Last but not least, we also thank the *invited speakers* who have greatly contributed to the success of our event:

- Mauricio Ayala-Rincón (UnB);
- Eduardo Bonelli (Stevens Institute of Technology);
- Delia Kesner (Université Paris Diderot and Institut Universitaire de France).

Cláudia Nalon
Giselle Reis

An Efficient Algorithm for Representing Piecewise Linear Functions into Logic

Sandro Preto[1,3] Marcelo Finger[1,2,4]

Institute of Mathematics and Statistics
University of São Paulo
São Paulo, Brazil

Abstract

Rational McNaughton functions may be implicitly represented by logical formulas in Łukasiewicz Infinitely-valued Logic by constraining the set of valuations to the ones that satisfy some specific formulas. This work investigates this implicit representation called representation modulo satisfiability and describes a polynomial algorithm that builds it — the representative formula and the constraining ones — for a given rational McNaughton function.

Keywords: Łukasiewicz Infinitely-valued Logic, Rational McNaughton Functions, Piecewise Linear Functions.

1 Introduction

The ability to represent any piecewise linear function with a logical formula allows us to apply automated reasoning techniques to the study of real systems, whose behavior is either modeled or approximated by such a function. However such an ability will only be effective if there are efficient ways to generate a formula in a target logic in which reasoning is not exceedingly complex. Classical logic with its binary semantics, despite of being a natural target for representing Boolean functions, may not be the natural way to represent continuous functions which inevitably would require some encoding of rational or real numbers; so we follow the path of electing some form of many-valued logic, whose semantics range over rational numbers, as a more adequate representation framework. That path has initially been explored by applications in fuzzy control [5].

Neural network interpretability is a challenge to the development of artificial intelligence and is also another motivation for the representation of piecewise linear functions, as described by [4]. In fact, a neural network, depending on its class of activation functions, can be seen either as a piecewise linear function or as a continuous function that can be approximated by one [12].

The first candidate to consider as a target logic is Łukasiewicz Infinitely-valued Logic ($Ł_\infty$), arguably one of the best studied many-valued logics [7]; it has several interesting properties such as continuous truth-functional semantics, classical logic as limit case, and well developed proof-theoretical and algebraic presentations. Its formulas are known to represent exactly the so-called McNaughton functions, consisting of $[0,1]$-valued piecewise linear functions *with integer coefficients* over $[0,1]^n$ [13,15]. This restriction to integer coefficients fails to fulfill, for instance, the hypotheses for the classic Stone-Weierstrass Approximation Theorem [16] and there is no known analogous to Proposition 1.1 below for McNaughton functions.

[1] This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

[2] Partly supported by Fapesp projects 2014/12236-1 and 2019/07665-4 and CNPq grant PQ 303609/2018-4.

[3] Email: spreto@ime.usp.br

[4] Email: mfinger@ime.usp.br

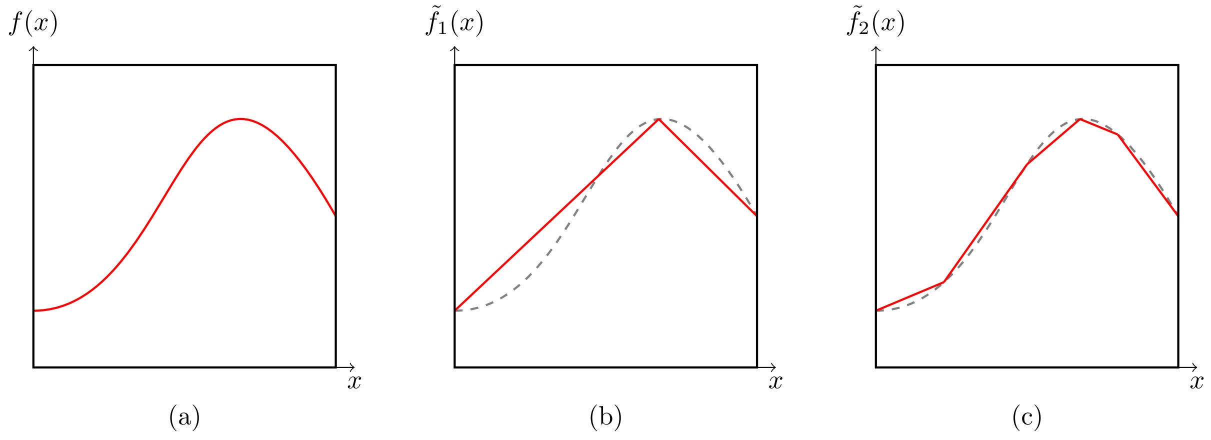


Fig. 1. Continuous one-variable function approximated by rational McNaughton functions.

This issue is circumvented by slightly modifying McNaughton functions to allow their linear pieces to have rational coefficients — the rational McNaughton functions; such generalization is enough to perform Weierstrass-like approximations [5,2]. Figure 1 shows a continuous function $f : [0,1] \to [0,1]$ (a) with two possible approximations by rational McNaughton functions: with two (b) and five different linear pieces (c). Note that continuous functions with more general domain and range might be normalized in order to perform such approximations.

Proposition 1.1 (Variation of Weierstrass Approximation Theorem [5]) *Let $f : [0,1]^n \to [0,1]$ be a continuous function and $\varepsilon > 0$. Then there is a rational McNaughton function $\tilde{f} : [0,1]^n \to [0,1]$ such that $|f(\mathbf{x}) - \tilde{f}(\mathbf{x})| < \varepsilon$, for all $\mathbf{x} \in [0,1]^n$.*

In this context, the target logic must be a system, preferably based on $Ł_\infty$, with semantics that comprehends all rational McNaughton functions and, given one such function, be endowed with an efficient algorithm that provides a formula that represents it. Furthermore, we highlight that it would be of little practical use if the reasoning complexity in such system is exceedingly high.

Esteva, Godo & Montagna propose logic $Ł\Pi\frac{1}{2}$ which extends $Ł_\infty$ with a product operator, its residuum, and a constant expressing the truth value $\frac{1}{2}$, not directly expressible in $Ł_\infty$ [8]. That logic not only allows for the expressivity of rational McNaughton functions but also expresses piecewise polynomials; as a consequence satisfiability over $Ł\Pi\frac{1}{2}$ requires finding roots of polynomials of n-degree rendering its complexity extremely high. Aguzzoli & Mundici propose logic $\exists Ł$ which also expresses rational McNaughton functions and has complexity Σ_2^p for the satisfiability problem [2,3]. Logic $\exists Ł$ extends $Ł_\infty$ and introduces rational numbers by providing restricted form of propositional quantification whose semantic counterpart is the maximization of a set of $Ł_\infty$-valuations of a formula.

Gerla introduces Rational Łukasiewicz Logic by extending $Ł_\infty$ with division operators δ_n that induces division by $n \in \mathbb{N}^*$ in its semantics [11]; its associated tautology problem is coNP-complete, which is a reasonable complexity for this task. This logic expresses all rational McNaughton functions however it was not provided an algorithm to build the representative formulas, and an attempt to derive one from the results in [11] would lead to the problem of representing McNaughton functions in $Ł_\infty$; it is known that this task may be done in polynomial time on the coefficients of some specific functions [1], however these methods lead to exponential time complexity if binary representation of the coefficients is used.

Finger & Preto provide a way to *implicitly* express rational McNaughton functions in $Ł_\infty$ called representation modulo satisfiability ($Ł_\infty$-MODSAT) [10]. For that, in addition to a representative formula, it is introduced a set of formulas that constrains $Ł_\infty$-valuations to those that satisfy all formulas in the set; a rational McNaughton function f is then represented by a pair $\langle \varphi, \Phi \rangle$, where φ is a formula that semantically acquires values $f(\mathbf{x})$, for $\mathbf{x} \in [0,1]^n$, from valuations in $\{v(\psi) = 1 \mid \psi \in \Phi\}$, where Φ is a set of formulas. Instead of an extension of the logic, this proposal works in $Ł_\infty$ itself, which has computational problems with reasonable complexity — e.g., satisfiability over $Ł_\infty$ is NP-complete [14]. Also, there already are available implementations of $Ł_\infty$-solvers which are discussed in the literature and for which phase transition phenomenon is identified [6,9]. Unfortunately, an attempt to derive a representation builder algorithm from results in [10] would also lead to an exponential blow up, since the proposed pairs for representing only truncated linear functions are already exponential in the binary representation of their coefficients.

Our goal here is to provide an efficient algorithm that, given a rational McNaughton function, outputs a pair $\langle \varphi, \Phi \rangle$ that represents it in the target system $Ł_\infty$-MODSAT. We show that all rational McNaughton functions may be represented modulo satisfiability by a constructive proof from which we derive a polynomial algorithm that builds such representation.

This paper is organized as follows: Section 2 introduces all necessary concepts of Łukasiewicz Infinitely-valued Logic and the definition of rational McNaughton functions; Section 3 has the formalization of the concept of representation modulo satisfiability; Section 4 provides a convention on rational McNaughton functions encoding for computation purposes as well as some results about these functions; Section 5 has a theoretical and algorithmic treatment of a particular case of representation modulo satisfiability of rational McNaughton functions which are truncated linear functions; and Section 6 finally treats, also theoretically and algorithmically, the representation modulo satisfiability of general rational McNaughton functions.

2 Preliminaries

The basic language $\mathcal{L}$ of Łukasiewicz Infinitely-valued Logic ($Ł_\infty$) comprehends the formulas built from a countable set of propositional variables $\mathbb{P}$, and disjunction ($\oplus$) and negation ($\neg$) operators. For the semantics, define a valuation as a function $v : \mathcal{L} \to [0, 1]$, such that, for $\varphi, \psi \in \mathcal{L}$:

$$v(\varphi \oplus \psi) = \min(1, v(\varphi) + v(\psi)); \tag{1}$$

$$v(\neg\varphi) = 1 - v(\varphi). \tag{2}$$

One may just give a function $v_\mathbb{P}$ which maps propositional variables to a value in the interval $[0, 1]$ and extend this function to a valuation by obeying (1) and (2). This extension is uniquely defined by such assignment to the variables in $\mathbb{P}$ given by $v_\mathbb{P}$.

We denote by **Val** the set of all valuations; by $\mathrm{Var}(\Phi)$ the set of all propositional variables occurring in the formulas $\varphi \in \Phi$; and by $\mathbf{X}_n$ the set of propositional variables $\{X_1, \dots, X_n\} \subset \mathbb{P}$. A formula φ is *satisfiable* if there exists a $v \in \mathbf{Val}$ such that $v(\varphi) = 1$; otherwise it is *unsatisfiable*. A set of formulas Φ is satisfiable if there exists a $v \in \mathbf{Val}$ such that $v(\varphi) = 1$, for all $\varphi \in \Phi$. We denote by $\mathbf{Val}_\Phi$ the set of all valuations $v \in \mathbf{Val}$ that satisfies a set of formulas Φ.

From disjunction and negation we derive the following operators:

Conjunction:	$\varphi \odot \psi =_{\mathrm{def}} \neg(\neg\varphi \oplus \neg\psi)$	$v(\varphi \odot \psi) = \max(0, v(\varphi) + v(\psi) - 1)$
Implication:	$\varphi \to \psi =_{\mathrm{def}} \neg\varphi \oplus \psi$	$v(\varphi \to \psi) = \min(1, 1 - v(\varphi) + v(\psi))$
Maximum:	$\varphi \vee \psi =_{\mathrm{def}} \neg(\neg\varphi \oplus \psi) \oplus \psi$	$v(\varphi \vee \psi) = \max(v(\varphi), v(\psi))$
Minimum:	$\varphi \wedge \psi =_{\mathrm{def}} \neg(\neg\varphi \vee \neg\psi)$	$v(\varphi \wedge \psi) = \min(v(\varphi), v(\psi))$
Bi-implication:	$\varphi \leftrightarrow \psi =_{\mathrm{def}} (\varphi \to \psi) \wedge (\psi \to \varphi)$	$v(\varphi \leftrightarrow \psi) = 1 - \lvert v(\varphi) - v(\psi) \rvert$

Note that $v(\varphi \to \psi) = 1$ iff $v(\varphi) \leq v(\psi)$; similarly, $v(\varphi \leftrightarrow \psi) = 1$ iff $v(\varphi) = v(\psi)$. Let X be a propositional variable, then, $v(X \odot \neg X) = 0$, for any $v \in \mathbf{Val}$; we define the constant $\mathbf{0}$ by $X \odot \neg X$. We also define $0\varphi =_{\mathrm{def}} \mathbf{0}$ and $n\varphi =_{\mathrm{def}} \varphi \oplus \dots \oplus \varphi$, n times, for $n \in \mathbb{N}^*$; and $\bigoplus_{i \in \varnothing} \varphi_i =_{\mathrm{def}} \mathbf{0}$.

Adapting the definition in [7], a *rational McNaughton function* $f : [0, 1]^n \to [0, 1]$ is a function that satisfies the following conditions:

- f is continuous with respect to the usual topology of $[0, 1]$ as an interval of the real number line;
- There are linear polynomials $p_1, \dots, p_m$ over $[0, 1]^n$ with rational coefficients such that, for each point $\mathbf{x} \in [0, 1]^n$, there is an index $i \in \{1, \dots, m\}$ with $f(\mathbf{x}) = p_i(\mathbf{x})$. Polynomials $p_1, \dots, p_m$ are the *linear pieces* of f.

3 Representation Modulo Φ-Satisfiable

A *McNaughton function* is a rational McNaughton function whose linear pieces have integer coefficients. Let φ be a $Ł_\infty$-formula with $\mathrm{Var}(\varphi) \subset \mathbf{X}_n$, we inductively associate to φ a function $f_\varphi : [0, 1]^n \to [0, 1]$ by:

(i) $f_{X_j}(x_1, \dots, x_n) = x_j$, for $j = 1, \dots, n$;

(ii) $f_{\neg\varphi}(x_1, \dots, x_n) = 1 - f_\varphi(x_1, \dots, x_n)$;

(iii) $f_{\varphi_1 \oplus \varphi_2}(x_1, \dots, x_n) = \min(1, f_{\varphi_1}(x_1, \dots, x_n) + f_{\varphi_2}(x_1, \dots, x_n))$.

We have that f_φ is a McNaughton function such that

$$f_\varphi(v(X_1), \ldots, v(X_n)) = v(\varphi), \text{ for } v \in \mathbf{Val}. \tag{3}$$

Reciprocally, McNaughton's Theorem [13] states that, for any McNaughton function f, there is a formula φ such that $f = f_\varphi$. We say that φ *represents* f.

Although formulas of $Ł_\infty$ only represent (integer) McNaughton functions, we take the strategy of restricting the set **Val** of valuations in order to implicitly represent rational McNaughton functions. For that, we start by noting that value of a formula φ according to some valuation v is determined only by the values associated to a finite set of propositional variables $\mathbf{X}$ such that $\text{Var}(\varphi) \subset \mathbf{X}$; this very property is the crux for the possibility that logical formulas represent functions. We next generalize this notion.

Definition 3.1 Let φ be a formula and let Φ be a set of formulas. We say that a set of variables $\mathbf{X}_n$ *determines* φ *modulo* Φ*-satisfiable* if:

- For any $\langle x_1, \ldots, x_n\rangle \in [0,1]^n$, there exists at least one valuation $v \in \mathbf{Val}_\Phi$, such that $v(X_j) = x_j$, for $j = 1, \ldots, n$;
- For any valuations $v, v' \in \mathbf{Val}_\Phi$, such that $v(X_j) = v'(X_j)$, for $j = 1, \ldots, n$, $v(\varphi) = v'(\varphi)$.

For instance, for any formula φ such that $\text{Var}(\varphi) \subset \mathbf{X}_n$, $\mathbf{X}_n$ determines φ modulo $\varnothing$-satisfiable, by truth functionality and the fact that $\mathbf{Val}_\varnothing = \mathbf{Val}$.

It is important to note that any set $\mathbf{X}_n$ can now represent a rational fraction $\frac{1}{d}$ by determining a propositional variable $Z_{\frac{1}{d}}$ modulo $\varphi_{\frac{1}{d}} = Z_{\frac{1}{d}} \leftrightarrow \neg(d-1)Z_{\frac{1}{d}}$ satisfiable, with $d \in \mathbb{N}^*$. In fact, for any valuation $v \in \mathbf{Val}$, if $v(\varphi_{\frac{1}{d}}) = 1$, then $v(Z_{\frac{1}{d}}) = \frac{1}{d}$. We define representation modulo satisfiability in a way that retrieves property (3).

Definition 3.2 Let $f : [0,1]^n \to [0,1]$ be a function, and $\langle\varphi, \Phi\rangle$ be a pair where φ is a formula and Φ is a set of formulas. We say that φ *represents* f *modulo* Φ*-satisfiable* or that $\langle\varphi, \Phi\rangle$ *represents* f *in the system* $Ł_\infty$*-MODSAT* if:

- $\mathbf{X}_n$ determines φ modulo Φ-satisfiable;
- $f(v(X_1), \ldots, v(X_n)) = v(\varphi)$, for $v \in \mathbf{Val}_\Phi$.

Representation modulo satisfiability presented in [10] has a different approach, which we call function-based and is more restrictive than the one presented here, which we call formula-based. However, the representation methods and algorithms we develop in this work apply to both approaches.

4 Rational McNaughton Functions

Our algorithm uses a lattice representation of rational McNaughton functions; before that we employ an encoding based in [17,18] as follows. Let Ω° be the interior of a set $\Omega \subset \mathbb{R}^n$. A rational McNaughton function $f : [0,1]^n \to [0,1]$ is given by m (not necessarily distinct) linear pieces

$$p_i(\mathbf{x}) = \gamma_{i0} + \gamma_{i1}x_1 + \cdots + \gamma_{in}x_n, \tag{4}$$

for $\mathbf{x} = \langle x_1, \ldots, x_n\rangle \in [0,1]^n$, $\gamma_{ij} \in \mathbb{Q}$ and $i = 1, \ldots, m$, with each linear piece p_i identical to f over a convex set $\Omega_i \subset [0,1]^n$ called *region* such that:

- $\bigcup_{i=1}^m \Omega_i = [0,1]^n$;
- $\Omega_{i'}^\circ \cap \Omega_{i''}^\circ = \varnothing$, for $i' \neq i''$;
- Regions Ω_i are given in such a way that there is a polynomial procedure to determine whether or not a linear piece p_k is *above* other linear piece p_i over region Ω_i, that is whether or not $p_k(\mathbf{x}) \geq p_i(\mathbf{x})$, for all $\mathbf{x} \in \Omega_i$.

A rational McNaughton function as above is said to be in *regional format.* Note that in this format the number of linear pieces is equal to the number of regions; in this case, the size of a function is the sum of the number of bits necessary to represent its linear pieces coefficients as fractions $\frac{a}{b}$ plus the space necessary for representing its regions in some assumed encoding.

Example 4.1 Rational McNaughton function f with graph in Figure 2 may be given by the linear pieces:

- $p_1(x_1, x_2) = \frac{4}{9} + \frac{2}{3}x_2$;
- $p_2(x_1, x_2) = \frac{5}{6} - \frac{1}{2}x_2$;

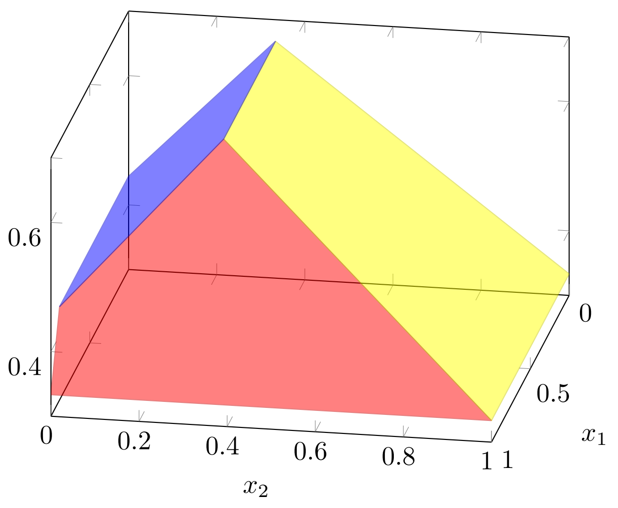


Fig. 2. Graph of rational McNaughton function with three linear pieces over $[0,1]^2$.

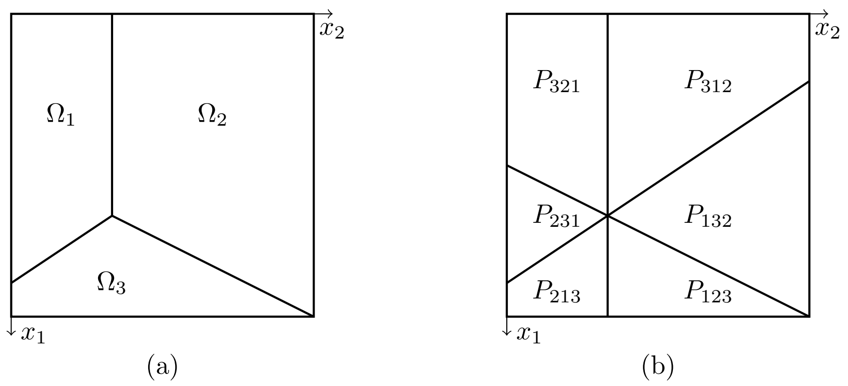


Fig. 3. Some possible region configurations for function f in Example 4.1.

Ω_1°	Ω_2°	Ω_3°
$8 - 9x_1 - 6x_2 > 0$	$1 - 2x_1 + x_2 > 0$	$-8 + 9x_1 + 6x_2 > 0$
$\frac{1}{3} - x_2 > 0$	$-\frac{1}{3} + x_2 > 0$	$-1 + 2x_1 - x_2 > 0$
$x_1 > 0$	$x_1 > 0$	$1 - x_1 > 0$
$x_2 > 0$	$1 - x_2 > 0$	$x_2 > 0$

Table 1
Sets Ω_i° for function f in Example 4.1.

- $p_3(x_1, x_2) = \frac{4}{3} - x_1$.

Regions associated to each linear piece are depicted in Figure 3(a). Determining if some linear piece p_k is above other linear piece p_i over Ω_i is equivalent to determining if the system of corresponding inequalities in Table 1 with added equation $p_k - p_i = 0$ has no solution and $p_k(\mathbf{x}_0) > p_i(\mathbf{x}_0)$, for some point $\mathbf{x}_0 \in \Omega_i^\circ$, a tractable process.

Let $f : [0,1]^n \to [0,1]$ be a rational McNaughton function as defined in Section 2, with distinct linear pieces $p_1, \dots, p_m$. For each permutation ρ of the set $\{1, \dots, m\}$, we define the polyhedron

$$P_\rho = \{\mathbf{x} \in [0,1]^n \mid p_{\rho(1)}(\mathbf{x}) \geq \cdots \geq p_{\rho(m)}(\mathbf{x})\}. \tag{5}$$

Let $\mathcal{C}$ be the set of n-dimensional polyhedra P_ρ, for some permutation ρ.

Proposition 4.2 *The set $\mathcal{C}$ has the following properties:*

(a) $\bigcup \mathcal{C} = [0,1]^n$*;*

(b) For polyhedron $P \in \mathcal{C}$ *and indexes* $i', i'' \in \{1, \ldots, m\}$ *with* $i' \neq i''$, $p_{i'}(\mathbf{x}) \neq p_{i''}(\mathbf{x})$, *for any* $\mathbf{x} \in P^\circ$*;*

(c) $P'^\circ \cap P''^\circ = \varnothing$, *for* $P', P'' \in \mathcal{C}$ *such that* $P' \neq P''$*;*

(d) For each polyhedron $P \in \mathcal{C}$, *there is an index* $i_P \in \{1, \ldots, m\}$ *such that* $f(\mathbf{x}) = p_{i_P}(\mathbf{x})$, *for* $\mathbf{x} \in [0,1]^n$.

Proof.

(a) For any $\mathbf{x} \in P \in \mathcal{C}$, $\mathbf{x} \in [0,1]^n$. On the other hand, for any $\mathbf{x} \in [0,1]^n$, there is a permutation ρ for which P_ρ is n-dimensional and $\mathbf{x} \in P_\rho$.

(b) Let $\mathbf{x} \in P^\circ$ and let $i', i'' \in \{1, \ldots, m\}$ be indexes such that $i' \neq i''$. Since $p_{i'}$ and $p_{i''}$ are linear pieces, if $p_{i'}(\mathbf{x}) = p_{i''}(\mathbf{x})$, for some $\mathbf{x} \in P^\circ$, there would be points $\mathbf{x}_1, \mathbf{x}_2 \in P^\circ$ in a neighborhood of $\mathbf{x}$ such that $p_{i'}(\mathbf{x}_1) < p_{i''}(\mathbf{x}_1)$ and $p_{i''}(\mathbf{x}_2) < p_{i'}(\mathbf{x}_2)$, contrary to the definition of P.

(c) Let $\mathbf{x} \in P'^\circ \cap P''^\circ$. Then, by definitions of P' and P'', there are $i', i'' \in \{1, \ldots, m\}$ such that $p_{i'}(\mathbf{x}) = p_{i''}(\mathbf{x})$, contrary to item (b).

(d) Let $\{i_1, \ldots, i_k\} \subset \{1, \ldots, m\}$ be a non-singleton set of indexes such that, for any $\mathbf{x} \in P^\circ$, there is $l \in \{1, \ldots, k\}$, such that $f(\mathbf{x}) = p_{i_l}(\mathbf{x})$ and $U_{i_l} = \{\mathbf{x} \in P^\circ \mid f(\mathbf{x}) = p_{i_l}(\mathbf{x})\} \neq \varnothing$, for $l = 1, \ldots, k$. We have that $\cup_{l=1}^k U_{i_l} = P^\circ$ and, by item (b), $U_{i_{l'}} \cap U_{i_{l''}} = \varnothing$, for $l' \neq l''$. As P° is a connected set, there are distinct $i', i'' \in \{i_1, \ldots, i_k\}$ and $\mathbf{b} \in P^\circ$ such that $\mathbf{b} \in \partial U_{i'}$ and $\mathbf{b} \in U_{i''}$. As $p_{i'}$ restricted to $U_{i'} \cup \{\mathbf{b}\}$ is continuous, for any sequence $\{\mathbf{b}_n\} \subset U_{i'}$ such that $\lim \mathbf{b}_n = \mathbf{b}$ (which exists since $\mathbf{b} \in \partial U_{i'}$), we have that $\lim f(\mathbf{b}_n) = \lim p_{i'}(\mathbf{b}_n) = p_{i'}(\mathbf{b})$. However, $f(\mathbf{b}) = p_{i''}(\mathbf{b}) \neq p_{i'}(\mathbf{b})$, by item (b), contrary to the continuity of f.

□

Polyhedra in $\mathcal{C}$ may play the role of regions since they are convex sets with the properties above; determining whether a linear piece p_k is above other linear piece p_i over $P \in \mathcal{C}$ comes down to comparing their values for some point $\mathbf{x} \in P^\circ$. Note that the same linear piece p_i may be associated to many distinct polyhedra. Thus, any rational McNaughton function may be encoded in regional format. Figure 3(b) shows the polyhedra-based configuration $\mathcal{C}$ for the function in Example 4.1.

The setback with describing a rational McNaughton function using the set $\mathcal{C}$ of polyhedra is that in the worst case $|\mathcal{C}| = m!$. However, in general there are smaller sets of regions that comply with representation restrictions above [18].

5 A Particular Case: Truncated Linear Functions

Let us show the possibility of representing a rational McNaughton function modulo satisfiability and develop a polynomial algorithm for computing such representation in the particular case that function is a truncated linear polynomial with rational coefficients.

Let $p : [0,1]^n \to \mathbb{R}$ be a nonzero linear polynomial given by

$$p(\mathbf{x}) = \frac{a_0}{b_0} + \frac{a_1}{b_1} x_1 + \cdots + \frac{a_n}{b_n} x_n, \tag{6}$$

for $\mathbf{x} = \langle x_1, \ldots, x_n \rangle \in [0,1]^n$, $a_j \in \mathbb{Z}$, and $b_j \in \mathbb{Z}_+^*$. We want to build a representation for the function $p^\# : [0,1]^n \to [0,1]$ given by

$$p^\#(\mathbf{x}) = \min\Big(1, \max\big(0, p(\mathbf{x})\big)\Big). \tag{7}$$

We have that $p^\#(\mathbf{x}) = 0$, if $p(\mathbf{x}) < 0$; $p^\#(\mathbf{x}) = 1$, if $p(\mathbf{x}) > 1$; and $p^\#(\mathbf{x}) = p(\mathbf{x})$, otherwise.

In order to rewrite expression (6), we define:

$$\begin{aligned}
&\alpha_j = a_j, \text{ for } j \in P;\\
&\alpha_j = -a_j, \text{ for } j \in N;\\
&\beta_j = \beta \cdot b_j, \text{ for } j = 0, \ldots, n;
\end{aligned}$$

where $j \in P$, if $a_j > 0$, and $j \in N$, if $a_j < 0$, with $P \cup N \subset \{0, \ldots, n\}$, and β is the least integer greater than or equal to

$$\max\left\{\sum_{j\in P}\frac{a_j}{b_j}, \quad -\sum_{j\in N}\frac{a_j}{b_j}\right\}.$$

We have that $\alpha_j \in \mathbb{Z}_+$ and $\beta_j \in \mathbb{Z}_+^*$, for $j = 0, \ldots, n$. Let $x_0 = 1$ and define functions $p_P : [0,1]^n \to \mathbb{R}$ and $p_N : [0,1]^n \to \mathbb{R}$, for $\mathbf{x} = \langle x_1, \ldots, x_n\rangle \in [0,1]^n$, by:

$$p_P(\mathbf{x}) = \sum_{j\in P}\frac{\alpha_j}{\beta_j}x_j; \qquad\qquad p_N(\mathbf{x}) = \sum_{j\in N}\frac{\alpha_j}{\beta_j}x_j. \tag{8}$$

Lemma 5.1 *Functions p, p_P, and p_N in (6) and (8) have the following properties, for $\mathbf{x} \in [0,1]^n$:*

(a) $p(\mathbf{x}) = \beta \cdot \big(p_P(\mathbf{x}) - p_N(\mathbf{x})\big)$;

(b) $0 \le p_P(\mathbf{x}), p_N(\mathbf{x}) \le 1$.

Proof. By elementary algebraic manipulation. □

Lemma above decomposes function p in terms of p_P and p_N, let us represent the latter ones. Let $Z_j^p, Z_{\frac{1}{\beta_j}} \in \mathbb{P}$. For a set of indexes $J \in \{P, N\}$, define:

$$\tilde{\varphi}_J = \bigoplus_{j\in J\setminus\{0\}} \alpha_j Z_j^p; \qquad\qquad \tilde{\Phi}_J = \bigcup_{j\in J\setminus\{0\}} \left\{\varphi_{\frac{1}{\beta_j}}, \quad \beta_j Z_j^p \leftrightarrow X_j, \quad Z_j^p \to Z_{\frac{1}{\beta_j}}\right\}.$$

And then, define:

$$\begin{array}{lll} \bar{\varphi}_J = \tilde{\varphi}_J; & \bar{\Phi}_J = \tilde{\Phi}_J, & \text{if } 0 \notin J; \\ \bar{\varphi}_J = \alpha_0 Z_{\frac{1}{\beta_0}} \oplus \tilde{\varphi}_J; & \bar{\Phi}_J = \tilde{\Phi}_J \cup \{\varphi_{\frac{1}{\beta_0}}\}, & \text{otherwise.} \end{array} \tag{9}$$

Lemma 5.2 *Functions p_P and p_N in (8) may respectively be represented by $\langle\bar{\varphi}_P, \bar{\Phi}_P\rangle$ and $\langle\bar{\varphi}_N, \bar{\Phi}_N\rangle$ in (9).*

Proof. Let $J \in \{P, N\}$. If $J = \varnothing$, then $\langle\bar{\varphi}_J, \bar{\Phi}_J\rangle = \langle\mathbf{0}, \varnothing\rangle$ represents p_J. For $\langle x_1, \ldots, x_n\rangle \in [0,1]^n$, define a valuation $v \in \mathbf{Val}$ such that $v(X_j) = x_j$ and $v(Z_j^p) = \frac{x_j}{\beta_j}$, for $j \in J \setminus \{0\}$, and $v(Z_{\frac{1}{\beta_j}}) = \frac{1}{\beta_j}$, for $j \in J$. We have that $v \in \mathbf{Val}_{\bar{\Phi}_J}$. Now, let $v, v' \in \mathbf{Val}_{\bar{\Phi}_J}$ such that $v(X_j) = v'(X_j)$, for $j = 1, \ldots, n$. By rational constant representation, $v(Z_{\frac{1}{\beta_j}}) = v'(Z_{\frac{1}{\beta_j}}) = \frac{1}{\beta_j}$, for $j \in J$. Thus $v(Z_j^p) \le \frac{1}{\beta_j}$ and $v'(Z_j^p) \le \frac{1}{\beta_j}$, which implies that $\beta_j \cdot v(Z_j^p) = v(\beta_j Z_j^p) = v(X_j) = v'(X_j) = v'(\beta_j Z_j^p) = \beta_j \cdot v'(Z_j^p)$ and, then, $v(Z_j^p) = v'(Z_j^p)$, for $j \in J \setminus \{0\}$. Therefore, $v(\bar{\varphi}_J) = v'(\bar{\varphi}_J)$ and $\mathbf{X}_n$ determines $\bar{\varphi}_J$ modulo $\bar{\Phi}_J$-satisfiable. Finally, suppose $v \in \mathbf{Val}_{\bar{\Phi}_J}$. In the case where $0 \in J$,

$$p_J(v(X_1), \ldots, v(X_n)) = \alpha_0 \cdot v(Z_{\frac{1}{\beta_0}}) + \sum_{j\in J\setminus\{0\}} \alpha_j \cdot v(Z_j^p) = v(\bar{\varphi}_J),$$

by Lemma 5.1 and aforementioned equations $v(Z_{\frac{1}{\beta_0}}) = \frac{1}{\beta_0}$ and $\beta_j \cdot v(Z_j^p) = v(X_j)$. The case where $0 \notin J$ is similar. □

For the final step towards a representation for $p^\#$, we define:

$$\bar{\varphi}_p = \beta[\neg(\bar{\varphi}_P \to \bar{\varphi}_N)]; \qquad\qquad \bar{\Phi}_p = \bar{\Phi}_P \cup \bar{\Phi}_N. \tag{10}$$

Theorem 5.3 *Function $p^\#$ in (7) may be represented by $\langle\bar{\varphi}_p, \bar{\Phi}_p\rangle$ in (10).*

Proof. For $\langle x_1, \ldots, x_n\rangle \in [0,1]^n$, there exists $v \in \mathbf{Val}_{\bar{\Phi}_p}$ such that $v(X_j) = x_j$ as in the proof of Lemma 5.2 with $J = P \cup N$. Now, let $v, v' \in \mathbf{Val}_{\bar{\Phi}_p}$ such that $v(X_j) = v'(X_j)$, for $j = 1, \ldots, n$. In particular, $v, v' \in \mathbf{Val}_{\bar{\Phi}_J}$ and, by Lemma 5.2, $v(\bar{\varphi}_J) = v'(\bar{\varphi}_J)$, for $J \in \{P, N\}$. Therefore, $v(\bar{\varphi}_p) = v'(\bar{\varphi}_p)$ and $\mathbf{X}_n$ determines $\bar{\varphi}_p$ modulo $\bar{\Phi}_p$-satisfiable. Finally, suppose $v \in \mathbf{Val}_{\bar{\Phi}_p}$. In particular, $v \in \mathbf{Val}_{\bar{\Phi}_P}$ and $v \in \mathbf{Val}_{\bar{\Phi}_N}$.

$\bar{\varphi}_{p_1}$:	$\neg\left(Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_2^{p_1} \oplus Z_2^{p_1} \to \mathbf{0}\right)$ $\oplus\neg\left(Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_2^{p_1} \oplus Z_2^{p_1} \to \mathbf{0}\right)$
$\bar{\Phi}_{p_1}$:	$Z_{\frac{1}{18}} \leftrightarrow \neg\Big(Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}}$ $\oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}} \oplus Z_{\frac{1}{18}}\Big)$ $Z_{\frac{1}{6}} \leftrightarrow \neg\left(Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}}\right)$ $Z_2^{p_1} \oplus Z_2^{p_1} \oplus Z_2^{p_1} \oplus Z_2^{p_1} \oplus Z_2^{p_1} \oplus Z_2^{p_1} \leftrightarrow X_2$ $Z_2^{p_1} \to Z_{\frac{1}{6}}$
$\bar{\varphi}_{p_2}$:	$\neg\left(Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \to Z_2^{p_2}\right)$
$\bar{\Phi}_{p_2}$:	$Z_{\frac{1}{6}} \leftrightarrow \neg\left(Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}}\right)$ $Z_{\frac{1}{2}} \leftrightarrow \neg Z_{\frac{1}{2}}$ $Z_2^{p_2} \oplus Z_2^{p_2} \leftrightarrow X_2$ $Z_2^{p_2} \to Z_{\frac{1}{2}}$
$\bar{\varphi}_{p_3}$:	$\neg\left(Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \to Z_1^{p_3}\right) \oplus \neg\left(Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \to Z_1^{p_3}\right)$
$\bar{\Phi}_{p_3}$:	$Z_{\frac{1}{6}} \leftrightarrow \neg\left(Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}} \oplus Z_{\frac{1}{6}}\right)$ $Z_{\frac{1}{2}} \leftrightarrow \neg Z_{\frac{1}{2}}$ $Z_1^{p_3} \oplus Z_1^{p_3} \leftrightarrow X_1$ $Z_1^{p_3} \to Z_{\frac{1}{2}}$

Table 2
Representations as in (10) for functions $p_1^\#$, $p_2^\#$ and $p_3^\#$, where functions p_1, p_2 and p_3 are from Example 4.1.

If $p(v(X_1),\ldots,v(X_n)) \leq 0$, by Lemma 5.1, $p_P(v(X_1),\ldots,v(X_n)) \leq p_N(v(X_1),\ldots,v(X_n))$. Therefore, by Lemma 5.2, $v(\bar{\varphi}_P) \leq v(\bar{\varphi}_N)$ and, then, $v(\bar{\varphi}_p) = 0$. On the other hand, if $p(v(X_1),\ldots,v(X_n)) \geq 0$, by Lemma 5.1, $p_P(v(X_1),\ldots,v(X_n)) \geq p_N(v(X_1),\ldots,v(X_n))$. Therefore, by Lemma 5.2, $v(\bar{\varphi}_P) \geq v(\bar{\varphi}_N)$ and, then, $v(\neg(\bar{\varphi}_P \to \bar{\varphi}_N)) = 1 - \min(1, 1 - v(\bar{\varphi}_P) + v(\bar{\varphi}_N)) = v(\bar{\varphi}_P) - v(\bar{\varphi}_N)$. Finally, by Lemmas 5.1 and 5.2, $p(v(X_1),\ldots,v(X_n)) = \beta \cdot (v(\bar{\varphi}_P) - v(\bar{\varphi}_N))$, hence $p^\#(v(X_1),\ldots,v(X_n)) = v(\bar{\varphi}_p)$ in any case. □

Table 2 shows how functions in Example 4.1 can be represented as in Theorem 5.3.

In order to set up a polynomial algorithm for computing a representation $\langle \varphi_p, \Phi_p \rangle$ for $p^\#$, we analyze more closely expressions $n\psi$, which show up in $\bar{\varphi}_p$ and in formulas in $\bar{\Phi}_p$. These expressions are exponential in the binary representation of n since it denotes an n-fold repetition of formula ψ. We deviate from this situation by using $\lfloor \log n \rfloor + 1$ new propositional variables $\xi_\psi^0, \xi_\psi^1, \ldots, \xi_\psi^{\lfloor \log n \rfloor}$ and replacing every occurrence of $n\psi$, where $n \in \mathbb{N} \setminus \{0, 1\}$, with the formula

$$\xi_{n\psi} =_{\text{def}} \bigoplus_{\substack{k=0 \\ n_k=1}}^{\lfloor \log n \rfloor} \xi_\psi^k, \tag{11}$$

where $n_k \in \{0,1\}$ comes from the binary representation $\sum_{k=0}^{\lfloor \log n \rfloor} 2^k n_k$ of n, and by adding the following formulas to $\bar{\Phi}_p$:

$$\begin{aligned} &\xi_\psi^0 \leftrightarrow \psi; \\ &\xi_\psi^k \leftrightarrow \xi_\psi^{k-1} \oplus \xi_\psi^{k-1}, \text{ for } k = 1, \dots, \lfloor \log n \rfloor. \end{aligned} \tag{12}$$

These formulas define the propositional variables ξ_ψ^k and we call $\Xi_{n\psi}$ the set that comprehends them. In this way we avoid exponential blow up as shown in Theorem 5.5.

Lemma 5.4 *Let $n \in \mathbb{N} \setminus \{0,1\}$, ψ be a formula, and $\xi_{n\psi}$ and $\Xi_{n\psi}$ be respectively a formula as in (11) and a set as in (12) built from n and ψ. For any valuation $v \in \mathbf{Val}_{\Xi_{n\psi}}$, $v(n\psi) = v(\xi_{n\psi})$.*

Proof. For $v \in \mathbf{Val}_{\Xi_{n\psi}}$ and $k = 0, \dots, \lfloor \log n \rfloor$, $v(\xi_\psi^k) = \min(1, 2^k v(\psi))$. Then,

$$\begin{aligned} v(n\psi) &= \min\left(1, \sum_{k=0}^{\lfloor \log n \rfloor} 2^k n_k v(\psi)\right) \\ &= \min\left(1, \sum_{k=0}^{\lfloor \log n \rfloor} v(\xi_\psi^k) n_k\right) = v\left(\bigoplus_{n_k=1} \xi_\psi^k\right) = v(\xi_{n\psi}), \end{aligned}$$

where $n_k \in \{0,1\}$ in the binary representation $n = \sum_{k=0}^{\lfloor \log n \rfloor} 2^k n_k$. □

Theorem 5.5 *Let $n \in \mathbb{N} \setminus \{0,1\}$, ψ be a formula, and $\langle \varphi_p, \Phi_p \rangle$ be a pair defined from representation $\langle \bar{\varphi}_p, \bar{\Phi}_p \rangle$ in (10) by replacing any occurrence of $n\psi$ in $\bar{\varphi}_p$ and $\bar{\Phi}_p$ with $\xi_{n\psi}$ in (11) and by adding formulas in set $\Xi_{n\psi}$ in (12) to $\bar{\Phi}_p$. Then, $\langle \varphi_p, \Phi_p \rangle$ is also a representation for $p^\#$ in (7). Furthermore, $\langle \varphi_p, \Phi_p \rangle$ is a representation for $p^\#$ even if it is defined by multiple suitable replacements of expressions $n_l \psi_l$, for $l = 1, \dots, L$.*

Proof. For $\langle x_1, \dots, x_n \rangle \in [0,1]^n$, define a valuation v such that $v(X_j) = x_j$ and $v(Z_j^p) = \frac{x_j}{\beta_j}$, for $j = 1, \dots, n$, $v(Z_{\frac{1}{\beta_j}}) = \frac{1}{\beta_j}$, for $j = 0, \dots, n$, $v(\xi_\psi^0) = v(\psi)$, and $v(\xi_\psi^k) = \min(1, v(\xi_\psi^{k-1}) + v(\xi_\psi^{k-1}))$, for $k = 1, \dots, \lfloor \log n \rfloor$. Note that $v \in \mathbf{Val}_{\bar{\Phi}_p}$ and $v \in \mathbf{Val}_{\Xi_{n\psi}}$, then, by Lemma 5.4, as $\Xi_{n\psi} \subset \Phi_p$, we have that $v \in \mathbf{Val}_{\Phi_p}$. Still, for any $v \in \Phi_p$, we have that $v \in \mathbf{Val}_{\Xi_{n\psi}}$ and, by Lemma 5.4, $v \in \bar{\Phi}_p$. Therefore, again by Lemma 5.4, for $v, v' \in \Phi_p$ such that $v(X_j) = v'(X_j)$, for $j = 1, \dots, n$, it follows that $v(\varphi_p) = v'(\varphi_p)$, $\mathbf{X}_n$ determines φ_p modulo Φ_p-satisfiable, and $p^\#(v(X_1), \dots, v(X_n)) = v(\varphi_p)$. This argument still holds when considering multiple replacements. □

We set $\langle \varphi_p, \Phi_p \rangle$ from $\langle \bar{\varphi}_p, \bar{\Phi}_p \rangle$ in (10) by properly replacing all occurrences of $n_l \psi_l$ as stated in the above theorem. By construction, $\langle \varphi_p, \Phi_p \rangle$ is given by

$$\varphi_p = \beta[\neg(\varphi_P \to \varphi_N)]; \qquad \Phi_p = \Phi_P \cup \Phi_N; \tag{13}$$

where φ_P, φ_N, Φ_P, and Φ_N are properly defined from their barred correspondents in (9). Table 3 shows how functions in Example 4.1 can be represented as in Theorem 5.5.

Algorithms 1 and 2 compute the representation modulo satisfiability of $n\psi$. Algorithm 1 returns $\mathbf{0}$ and ψ in the limit cases $n = 0$ and $n = 1$ (lines 1 to 5); when $n \in \mathbb{N} \setminus \{0,1\}$, it returns formula $\xi_{n\psi}$ in (11) by building it in line 6 plus a $\lfloor \log n \rfloor + 1$ iteration loop (lines 7 to 13) where the n_k's in the binary representation of n are calculated by the routine in lines 8 and 9. Algorithm 2 returns $\varnothing$ in the limit cases $n = 0$ and $n = 1$ (lines 1 to 3); when $n \in \mathbb{N} \setminus \{0,1\}$, it returns set $\Xi_{n\psi}$ that comprehends formulas (12) by building it in line 4 plus a $\lfloor \log n \rfloor$ iteration loop (lines 5 to 7). Both algorithms terminate in time $O(\log n)$ assuming propositional variables are all represented with a constant size.

Algorithm 3 computes a representation modulo satisfiability for $p^\#$. It returns $\langle \mathbf{0}, \varnothing \rangle$ in the limit case $a_0 = \cdots = a_n = 0$ (lines 1 to 3); otherwise it returns representation $\langle \varphi_p, \Phi_p \rangle$ given in (13). From line 4 to line 15, the algorithm sets all P, N, α_j, β_j, and β, for $j = 0, \dots, n$, which are used to rewrite function p in terms of p_P and p_N as in Lemma 5.1. From line 16 to line 26, it writes formulas φ_P and φ_N and adds formulas in Φ_P and Φ_N to Φ_p. For $J \in \{P, N\}$, it works throughout a $|J|$ iteration loop where each iteration takes a coefficient

φ_{p_1}:	$\xi^1_{\neg(\xi^2_{Z_{\frac{1}{18}}} \oplus \xi^1_{Z_2^{p_1}} \to \mathbf{0})}$	
Φ_{p_1}:	$Z_{\frac{1}{18}} \leftrightarrow \neg\left(\xi^4_{Z_{\frac{1}{18}}} \oplus \xi^0_{Z_{\frac{1}{18}}}\right)$	$\xi^0_{Z_2^{p_1}} \leftrightarrow Z_2^{p_1}$
	$\xi^0_{Z_{\frac{1}{18}}} \leftrightarrow Z_{\frac{1}{18}}$	$\xi^1_{Z_2^{p_1}} \leftrightarrow \xi^0_{Z_2^{p_1}} \oplus \xi^0_{Z_2^{p_1}}$
	$\xi^1_{Z_{\frac{1}{18}}} \leftrightarrow \xi^0_{Z_{\frac{1}{18}}} \oplus \xi^0_{Z_{\frac{1}{18}}}$	$\xi^2_{Z_2^{p_1}} \leftrightarrow \xi^1_{Z_2^{p_1}} \oplus \xi^1_{Z_2^{p_1}}$
	$\xi^2_{Z_{\frac{1}{18}}} \leftrightarrow \xi^1_{Z_{\frac{1}{18}}} \oplus \xi^1_{Z_{\frac{1}{18}}}$	$\xi^0_{Z_{\frac{1}{6}}} \leftrightarrow Z_{\frac{1}{6}}$
	$\xi^3_{Z_{\frac{1}{18}}} \leftrightarrow \xi^2_{Z_{\frac{1}{18}}} \oplus \xi^2_{Z_{\frac{1}{18}}}$	$\xi^1_{Z_{\frac{1}{6}}} \leftrightarrow \xi^0_{Z_{\frac{1}{6}}} \oplus \xi^0_{Z_{\frac{1}{6}}}$
	$\xi^4_{Z_{\frac{1}{18}}} \leftrightarrow \xi^3_{Z_{\frac{1}{18}}} \oplus \xi^3_{Z_{\frac{1}{18}}}$	$\xi^2_{Z_{\frac{1}{6}}} \leftrightarrow \xi^1_{Z_{\frac{1}{6}}} \oplus \xi^1_{Z_{\frac{1}{6}}}$
	$Z_{\frac{1}{6}} \leftrightarrow \neg\left(\xi^2_{Z_{\frac{1}{6}}} \oplus \xi^0_{Z_{\frac{1}{6}}}\right)$	$\xi^0_{\neg(\xi^2_{Z_{\frac{1}{18}}} \oplus \xi^1_{Z_2^{p_1}} \to \mathbf{0})} \leftrightarrow \neg\left(\xi^2_{Z_{\frac{1}{18}}} \oplus \xi^1_{Z_2^{p_1}} \to \mathbf{0}\right)$
	$\xi^2_{Z_2^{p_1}} \oplus \xi^1_{Z_2^{p_1}} \leftrightarrow X_2$	$\xi^1_{\neg(\xi^2_{Z_{\frac{1}{18}}} \oplus \xi^1_{Z_2^{p_1}} \to \mathbf{0})} \leftrightarrow \xi^0_{\neg(\xi^2_{Z_{\frac{1}{18}}} \oplus \xi^1_{Z_2^{p_1}} \to \mathbf{0})} \oplus \xi^0_{\neg(\xi^2_{Z_{\frac{1}{18}}} \oplus \xi^1_{Z_2^{p_1}} \to \mathbf{0})}$
	$Z_2^{p_1} \to Z_{\frac{1}{6}}$	
φ_{p_2}:	$\neg\left(\xi^2_{Z_{\frac{1}{6}}} \oplus \xi^0_{Z_{\frac{1}{6}}} \to Z_2^{p_2}\right)$	
Φ_{p_2}:	$Z_{\frac{1}{6}} \leftrightarrow \neg\left(\xi^2_{Z_{\frac{1}{6}}} \oplus \xi^0_{Z_{\frac{1}{6}}}\right)$	$\xi^1_{Z_{\frac{1}{6}}} \leftrightarrow \xi^0_{Z_{\frac{1}{6}}} \oplus \xi^0_{Z_{\frac{1}{6}}}$
	$Z_{\frac{1}{2}} \leftrightarrow \neg Z_{\frac{1}{2}}$	$\xi^2_{Z_{\frac{1}{6}}} \leftrightarrow \xi^1_{Z_{\frac{1}{6}}} \oplus \xi^1_{Z_{\frac{1}{6}}}$
	$\xi^1_{Z_2^{p_2}} \leftrightarrow X_2$	$\xi^0_{Z_2^{p_2}} \leftrightarrow Z_2^{p_2}$
	$Z_2^{p_2} \to Z_{\frac{1}{2}}$	$\xi^1_{Z_2^{p_2}} \leftrightarrow \xi^0_{Z_2^{p_2}} \oplus \xi^0_{Z_2^{p_2}}$
	$\xi^0_{Z_{\frac{1}{6}}} \leftrightarrow Z_{\frac{1}{6}}$	
φ_{p_3}:	$\xi^1_{\neg(\xi^2_{Z_{\frac{1}{6}}} \to Z_1^{p_3})}$	
Φ_{p_3}:	$Z_{\frac{1}{6}} \leftrightarrow \neg\left(\xi^2_{Z_{\frac{1}{6}}} \oplus \xi^0_{Z_{\frac{1}{6}}}\right)$	$Z_1^{p_3} \to Z_{\frac{1}{2}}$
	$\xi^0_{Z_{\frac{1}{6}}} \leftrightarrow Z_{\frac{1}{6}}$	$\xi^0_{Z_1^{p_3}} \leftrightarrow Z_1^{p_3}$
	$\xi^1_{Z_{\frac{1}{6}}} \leftrightarrow \xi^0_{Z_{\frac{1}{6}}} \oplus \xi^0_{Z_{\frac{1}{6}}}$	$\xi^1_{Z_1^{p_3}} \leftrightarrow \xi^0_{Z_1^{p_3}} \oplus \xi^0_{Z_1^{p_3}}$
	$\xi^2_{Z_{\frac{1}{6}}} \leftrightarrow \xi^1_{Z_{\frac{1}{6}}} \oplus \xi^1_{Z_{\frac{1}{6}}}$	$\xi^0_{\neg(\xi^2_{Z_{\frac{1}{6}}} \to Z_1^{p_3})} \leftrightarrow \neg\left(\xi^2_{Z_{\frac{1}{6}}} \to Z_1^{p_3}\right)$
	$Z_{\frac{1}{2}} \leftrightarrow \neg Z_{\frac{1}{2}}$	$\xi^1_{\neg(\xi^2_{Z_{\frac{1}{6}}} \to Z_1^{p_3})} \leftrightarrow \xi^0_{\neg(\xi^2_{Z_{\frac{1}{6}}} \to Z_1^{p_3})} \oplus \xi^0_{\neg(\xi^2_{Z_{\frac{1}{6}}} \to Z_1^{p_3})}$
	$\xi^1_{Z_1^{p_3}} \leftrightarrow X_1$	

Table 3
Representations as in (13) for functions $p_1^{\#}$, $p_2^{\#}$ and $p_3^{\#}$, where functions p_1, p_2 and p_3 are from Example 4.1.

Algorithm 1 BINARY-F: computes formula $\xi_{n\psi}$ in (11) or $\mathbf{0}$ or ψ

```
Input: A natural number $n$ and a formula $\psi$.
Output: Formula $\xi_{n\psi}$.
 1: if $n = 0$ then
 2:     return $\mathbf{0}$;
 3: else if $n = 1$ then
 4:     return $\psi$;
 5: end if
 6: $q := n$, $n_k := 0$, $\xi_{n\psi} := \mathbf{0}$;
 7: for $k = 0, \dots, \lfloor \log n \rfloor$ do
 8:     $n_k :=$ remainder from division of $q$ by 2;
 9:     $q :=$ quotient from division of $q$ by 2;
10:     if $n_k = 1$ then
11:         $\xi_{n\psi} := \xi^k_\psi \oplus \xi_{n\psi}$;
12:     end if
13: end for
14: return $\xi_{n\psi}$;
```

Algorithm 2 BINARY-S: computes set $\Xi_{n\psi}$ in (12) or $\varnothing$

```
Input: A natural number $n$ and a formula $\psi$.
Output: Set $\Xi_{n\psi}$.
 1: if $n = 0$ or $n = 1$ then
 2:     return $\varnothing$;
 3: end if
 4: $\Xi_{n\psi} := \{\xi^0_\psi \leftrightarrow \psi\}$;
 5: for $k = 1, \dots, \lfloor \log n \rfloor$ do
 6:     $\Xi_{n\psi} := \Xi_{n\psi} \cup \{\xi^k_\psi \leftrightarrow \xi^{k-1}_\psi \oplus \xi^{k-1}_\psi\}$;
 7: end for
 8: return $\Xi_{n\psi}$;
```

$\frac{a_j}{b_j}$ into account, where it treats $\frac{a_0}{b_0}$ (lines 18 to 21) separately from the others (lines 22 to 25). In lines 27 and 28 it finally writes formula φ_p and completes set Φ_p.

Theorem 5.6 *Given a rational linear function p by its coefficients, a representation $\langle \varphi_p, \Phi_p \rangle$ for $p^\#$ may be computed in polynomial time by Algorithm 3.*

Proof. Algorithm 3 builds representation $\langle \varphi_p, \Phi_p \rangle$ in (13). So, its correctness follows from Theorem 5.5. Let $[0,1]^n$ be the domain of p and M the maximum size of a binary representation for numbers among a_j and b_j; then the input size of p is at most $2(n+1)M$. The algorithm first calculates in polynomial time all β, α_j and β_j; let μ be the maximum size of a binary representation for numbers among β, α_j and β_j. Then, it proceeds to writing the representation which is made up of at most $3(n+1)$ propositional variables of the type X_j, Z^p_j and $Z_{\frac{1}{\beta_j}}$, and $2(n+1)\mu + \mu$ propositional variables of the type ξ^k_ψ, a quantity polynomially proportional to the size of the input. Thus, the size of the representation for each propositional variable may be assumed to be a constant π also polynomially proportional to the size of the input. Next, the algorithm calculates formulas φ_P and φ_N and sets Φ_P and Φ_N in $n+1$ steps; in each one it calculates the part associated to a coefficient $\frac{\alpha_i}{\beta_i}$. For each part, computation takes polynomial time on π and at most three executions of routines BINARY-F (Algorithm 2) and BINARY-S (Algorithm 1) with argument $\langle \nu, P \rangle$, where ν is α_i, β_i or $\beta_i - 1$, which are already or may be quickly computed, and P is a propositional variable. In these cases BINARY-F and BINARY-S run in polynomial time on μ and π. The algorithm finishes calculating φ_p and Φ_p by running BINARY-F and BINARY-S with argument $\langle \beta, \neg(\varphi_P \to \varphi_N) \rangle$. Now, BINARY-F runs in polynomial time on μ and π and BINARY-S runs in polynomial time on μ, π and the size of $\neg(\varphi_P \to \varphi_N)$. After all, Algorithm 4 terminates in polynomial time. □

We call REPRESENT-TL-F and REPRESENT-TL-S the routines that separately compute φ_p and Φ_p, respectively. Both may be easily derived from routine REPRESENT-TL in Algorithm 3.

Algorithm 3 REPRESENT-TL: computing representations for truncated linear functions

Input: A linear function p given by its rational coefficients $\frac{a_0}{b_0}, \frac{a_1}{b_1}, \dots, \frac{a_n}{b_n}$.

Output: A representation $\langle \varphi_p, \Phi_p \rangle$ for the truncated function $p^{\#}$.

```
1: if $a_1 = \cdots = a_n = 0$ then
2:   return $\langle \mathbf{0}, \varnothing \rangle$;
3: end if
4: $P := \varnothing$, $N := \varnothing$;
5: for $j := 0, \dots, n$ do
6:   if $a_j > 0$ then
7:     $P := P \cup \{j\}$, $\alpha_j := a_j$;
8:   else if $a_j < 0$ then
9:     $N := N \cup \{j\}$, $\alpha_j := -a_j$;
10:  end if
11: end for
12: $\beta :=$ least integer greater than or equal to $\max\{\sum_{j \in P} \frac{a_j}{b_j}, \quad -\sum_{j \in N} \frac{a_j}{b_j}\}$;
13: for $j \in P \cup N$ do
14:   $\beta_j := \beta \cdot b_j$;
15: end for
16: $\varphi_P := \mathbf{0}$, $\varphi_N := \mathbf{0}$, $\Phi_p := \varnothing$;
17: for $J = P, N$ do
18:   if $0 \in J$ then
19:     $\varphi_J := \varphi_J \oplus$ BINARY-F$(\alpha_0, Z_{\frac{1}{\beta_0}})$;
20:     $\Phi_p := \Phi_p \cup \{Z_{\frac{1}{\beta_0}} \leftrightarrow \neg$BINARY-F$(\beta_0 - 1, Z_{\frac{1}{\beta_0}})\} \cup$ BINARY-S$(\alpha_0, Z_{\frac{1}{\beta_0}}) \cup$ BINARY-S$(\beta_0 - 1, Z_{\frac{1}{\beta_0}})$;
21:   end if
22:   for $j \in J \setminus \{0\}$ do
23:     $\varphi_J := \varphi_J \oplus$ BINARY-F$(\alpha_j, Z_j^p)$;
24:     $\Phi_p := \Phi_p \cup \{Z_{\frac{1}{\beta_j}} \leftrightarrow \neg$BINARY-F$(\beta_j - 1, Z_{\frac{1}{\beta_j}})$, BINARY-F$(\beta_j, Z_j^p) \leftrightarrow X_j$, $Z_j^p \to Z_{\frac{1}{\beta_j}}\} \cup$
        BINARY-S$(\alpha_j, Z_j^p) \cup$ BINARY-S$(\beta_j - 1, Z_{\frac{1}{\beta_j}}) \cup$ BINARY-S$(\beta_j, Z_j^p)$;
25:   end for
26: end for
27: $\varphi_p :=$ BINARY-F$(\beta, \neg(\varphi_P \to \varphi_N))$;
28: $\Phi_p := \Phi_p \cup$ BINARY-S$(\beta, \neg(\varphi_P \to \varphi_N))$;
29: return $\langle \varphi_p, \Phi_p \rangle$;
```

6 The General Case

Given a rational McNaughton function formatted as in Section 4, we now compute a logical representation for it. Let $f : [0,1]^n \to [0,1]$ be a rational McNaughton function in regional format with linear pieces:

$$p_i(\mathbf{x}) = \frac{a_{i0}}{b_{i0}} + \frac{a_{i1}}{b_{i1}} x_1 + \cdots + \frac{a_{in}}{b_{in}} x_n, \tag{14}$$

for $\mathbf{x} = \langle x_1, \dots, x_n \rangle \in [0,1]^n$, $a_{ij} \in \mathbb{Z}$, $b_{ij} \in \mathbb{Z}_+^*$ and $i = 1, \dots, m$, with each piece identical to f in region Ω_i, for $i = 1, \dots, m$. We call ABOVE(p_k,p_i) the polynomial routine that decides if linear piece p_k is above a different linear piece p_i over Ω_i.

Let $\langle \varphi_{p_i}, \Phi_{p_i} \rangle$ be the representation for $p_i^{\#}$ given by Theorem 5.5, for $i = 1, \dots, m$. We define:

$$\varphi = \bigvee_{i=1}^{m} \varphi_{\Omega_i}, \text{ with } \varphi_{\Omega_i} = \bigwedge_{k \in K} \varphi_{p_k}; \qquad \Phi = \bigcup_{i=1}^{m} \Phi_{p_i}; \tag{15}$$

where $k \in K$ if $p_k(\mathbf{x}) \geq p_i(\mathbf{x})$, for all $\mathbf{x} \in \Omega_i$. We are able to state the following representation result which is adapted from [17,18].

Lemma 6.1 *Let f be a rational McNaughton function in regional format with linear pieces given by (14), and let φ_{Ω_i} be a formula and Φ a set as in (15). Then, $v(\varphi_{\Omega_i}) \leq f(v(X_1), \dots, v(X_n))$, for $v \in \mathbf{Val}_\Phi$ and $i = 1, \dots, m$.*

φ:	$(\varphi_{p_1} \wedge \varphi_{p_2} \wedge \varphi_{p_3}) \vee (\varphi_{p_1} \wedge \varphi_{p_2} \wedge \varphi_{p_3}) \vee (\varphi_{p_1} \wedge \varphi_{p_2} \wedge \varphi_{p_3})$
Φ:	$\Phi_{p_1} \cup \Phi_{p_2} \cup \Phi_{p_3}$

Table 4
Representation as in (15) for function f from Example 4.1.

Proof. Let $v \in \mathbf{Val}_\Phi$ and $\mathbf{x}_0 = \langle v(X_1), \ldots, v(X_n)\rangle$. In particular, $v \in \mathbf{Val}_{\Phi_{p_i}}$, for $i \in K$ and, by Theorem 5.5,

$$v(\varphi_{\Omega_i}) = \min_{k \in K} p_k^{\#}(\mathbf{x}_0).$$

If $\mathbf{x}_0 \in \Omega_i$, then $v(\varphi_{\Omega_i}) \leq p_i^{\#}(\mathbf{x}_0) = p_i(\mathbf{x}_0) = f(\mathbf{x}_0)$. On the other hand, if $\mathbf{x}_0 \notin \Omega_i$, there is some k_0 such that $\mathbf{x}_0 \in \Omega_{k_0}$. In the case $p_{k_0}(\mathbf{x}) \geq p_i(\mathbf{x})$, for all $\mathbf{x} \in \Omega_i$, then $k_0 \in K$ and $v(\varphi_{\Omega_i}) \leq p_{k_0}^{\#}(\mathbf{x}_0) = p_{k_0}(\mathbf{x}_0) = f(\mathbf{x}_0)$. In the case there is $\mathbf{x}' \in \Omega_i$ such that $p_{k_0}(\mathbf{x}') < p_i(\mathbf{x}')$, continuity of f yields that there is $t \in K$ such that $p_t(\mathbf{x}) \geq p_i(\mathbf{x})$, for all $\mathbf{x} \in \Omega_i$ and $p_t(\mathbf{x}) \leq p_{k_0}(\mathbf{x})$, for all $\mathbf{x} \in \Omega_{k_0}$. Therefore, $v(\varphi_{\Omega_i}) \leq p_t^{\#}(\mathbf{x}_0) \leq p_t(\mathbf{x}_0) \leq p_{k_0}(\mathbf{x}_0) = p_{k_0}^{\#}(\mathbf{x}_0) = f(\mathbf{x}_0)$. □

Theorem 6.2 *Any rational McNaughton function may be represented by $\langle \varphi, \Phi\rangle$ in (15).*

Proof. First note that any rational McNaughton function may be put in regional format as showed in Section 4. For $\langle x_1, \ldots, x_n\rangle \in [0,1]^n$, define a valuation $v \in \mathbf{Val}_\Phi$ such that $v(X_j) = x_j$ and $v(Z_j^{p_i}) = \frac{x_j}{\beta_{ij}}$, for $i = 1, \ldots, m, j = 1, \ldots, n$, $v(Z_{\frac{1}{\beta_{ij}}}) = \frac{1}{\beta_{ij}}$, for $i = 1, \ldots, m, j = 0, \ldots, n$, $v(\xi_\psi^0) = v(\psi)$, and $v(\xi_\psi^k) = \min(1, v(\xi_\psi^{k-1}) + v(\xi_\psi^{k-1}))$, for $k = 1, \ldots, \lfloor \log n \rfloor$, for any $n\psi$ that occurs in φ and Φ. Now, let $v, v' \in \mathbf{Val}_\Phi$ such that $v(X_j) = v'(X_j)$, for $j = 1, \ldots, n$. In particular, $v, v' \in \mathbf{Val}_{\Phi_{p_i}}$, for $i = 1, \ldots, m$, and, by Theorem 5.5, $v(\varphi_{p_i}) = v'(\varphi_{p_i})$, for $i = 1, \ldots, m$. Therefore, $v(\varphi) = v'(\varphi)$ and $\mathbf{X}_n$ determines φ modulo Φ-satisfiable. Finally, suppose $v \in \mathbf{Val}_\Phi$. There is some $k_0 \in K$ such that $\langle v(X_1), \ldots, v(X_n)\rangle \in \Omega_{k_0}$. Note that $v(\varphi_{\Omega_{k_0}}) = f(v(X_1), \ldots, v(X_n))$. Therefore,

$$f(v(X_1), \ldots, v(X_n)) = \max_{i=1,\ldots,m} v(\varphi_{\Omega_i}) = v(\varphi_{\Omega_{k_0}}),$$

by Lemma 6.1. □

Table 4 shows how function f in Example 4.1 can be represented as in Theorem 6.2.

Algorithm 4 REPRESENT: computing representations for rational McNaughton functions

Input: A rational McNaughton function f in regional format given by its linear pieces coefficients $\frac{a_{10}}{b_{10}}, \ldots, \frac{a_{1n}}{b_{1n}}, \ldots, \frac{a_{m0}}{b_{m0}}, \ldots, \frac{a_{mn}}{b_{mn}}$ and regions $\Omega_1, \ldots, \Omega_m$.
Output: A representation $\langle \varphi, \Phi\rangle$ for the rational McNaughton function f.

```
1:  $\Phi := \varnothing$;
2:  for $i = 1, \ldots, m$ do
3:    $\varphi_{p_i} :=$ REPRESENT-TL-F($\frac{a_{i0}}{b_{i0}}, \ldots, \frac{a_{in}}{b_{in}}$);
4:    $\varphi_{\Omega_i} := \varphi_{p_i}$;
5:  end for
6:  for $i = 1, \ldots, m$ do
7:    for $k = 1, \ldots, i-1, i+1, \ldots, m$ do
8:      if ABOVE($p_k, p_i$) = true then
9:        $\varphi_{\Omega_i} = \varphi_{\Omega_i} \wedge \varphi_{p_k}$;
10:     end if
11:   end for
12:   $\Phi := \Phi \cup$ REPRESENT-TL-S($\frac{a_{i0}}{b_{i0}}, \ldots, \frac{a_{in}}{b_{in}}$);
13: end for
14: $\varphi := \varphi_{\Omega_1} \vee \cdots \vee \varphi_{\Omega_m}$;
15: return $\langle \varphi, \Phi\rangle$;
```

Algorithm 4 returns representation $\langle \varphi, \Phi \rangle$ for function f with linear pieces given in (14). From line 1 to line 13, the algorithm writes formulas φ_{Ω_i} and the set Φ: it first computes formulas φ_{p_i} (lines 2 to 5) by means of routine REPRESENT-TL-F and then it writes φ_{Ω_i} (lines 7 to 11) by means of routine ABOVE. It writes set Φ computing each Φ_{p_i} by means of routine REPRESENT-TL-S (line 12). In line 14 it writes formula φ.

Theorem 6.3 *Given a rational McNaughton function f in regional format, a logical representation for it may be computed in polynomial time on the size of f by Algorithm 4.*

Proof. Algorithm 4 builds representation $\langle \varphi, \Phi \rangle$ in (15). So, the algorithm correctness follows from Theorem 6.2. The size of f is the space necessary to storage the coefficients of its m linear pieces $p_1, \dots, p_m$ and the regions $\Omega_1, \dots, \Omega_m$. The algorithm first calculates m representative formulas φ_{p_i} by REPRESENT-TL-F, which takes polynomial time on the size of f by Theorem 5.6. Then, it builds formulas φ_{Ω_i} from the already built representative formulas in m^2 steps; in each of these steps it runs routine ABOVE in assumed polynomial time. Along with the above computation, the algorithm also builds set Φ in m steps; in each one it calculates set Φ_{p_i} by REPRESENT-TL-S, which takes polynomial time on the size of f by Theorem 5.6. Finally, the algorithm calculates φ from formulas φ_{Ω_i} already computed. After all, Algorithm 4 terminates in polynomial time. □

7 Conclusions

We introduced a way to represent functions by logical formulas in Łukasiewicz Infinitely-valued Logic — the representation modulo satisfiability —, and we showed by a constructive proof that all rational McNaughton functions can be represented this way. Moreover, we derive an algorithm that builds such a representation in polynomial time on the size of the function. For the future, we hope to couple this algorithm with algorithms that approximate (normalized) continuous functions by rational McNaughton functions; also, apply these approximations to the study of real systems such as neural networks through automated reasoning techniques.

References

[1] Aguzzoli, S., *The complexity of McNaughton functions of one variable*, Advances in Applied Mathematics **21** (1998), pp. 58–77.

[2] Aguzzoli, S. and D. Mundici, *Weierstrass approximations by Łukasiewicz formulas with one quantified variable*, in: *Proceedings 31st IEEE International Symposium on Multiple-Valued Logic*, IEEE, 2001, pp. 361–366.

[3] Aguzzoli, S. and D. Mundici, "Weierstrass Approximation Theorem and Łukasiewicz Formulas with one Quantified Variable," Physica-Verlag HD, Heidelberg, 2003 pp. 315–335.

[4] Amato, P., A. Di Nola and B. Gerla, *Neural networks and rational Łukasiewicz logic*, in: *2002 Annual Meeting of the North American Fuzzy Information Processing Society Proceedings. NAFIPS-FLINT 2002 (Cat. No. 02TH8622)*, IEEE, 2002, pp. 506–510.

[5] Amato, P. and M. Porto, *An algorithm for the automatic generation of a logical formula representing a control law*, Neural Network World **10** (2000), pp. 777–786.

[6] Bofill, M., F. Manya, A. Vidal and M. Villaret, *Finding hard instances of satisfiability in Lukasiewicz logics*, in: *Multiple-Valued Logic (ISMVL), 2015 IEEE International Symposium on*, IEEE, 2015, pp. 30–35.

[7] Cignoli, R., I. D'Ottaviano and D. Mundici, "Algebraic Foundations of Many-Valued Reasoning," Trends in Logic, Springer Netherlands, 2000.

[8] Esteva, F., L. Godo and F. Montagna, *The $L\Pi$ and $L\Pi\frac{1}{2}$ logics: two complete fuzzy systems joining Lukasiewicz and product logics*, Archive for Mathematical Logic **40** (2001), pp. 39–67.

[9] Finger, M. and S. Preto, *Probably half true: Probabilistic satisfiability over Lukasiewicz infinitely-valued logic*, in: D. Galmiche, S. Schulz and R. Sebastiani, editors, *Automated Reasoning* (2018), pp. 194–210.

[10] Finger, M. and S. Preto, *Probably partially true: Satisfiability for Lukasiewicz infinitely-valued probabilistic logic and related topics*, Journal of Automated Reasoning (2020).
URL https://doi.org/10.1007/s10817-020-09558-9

[11] Gerla, B., *Rational Lukasiewicz logic and DMV-algebras*, Neural Network World **11** (2001), pp. 579–594.

[12] Leshno, M., V. Y. Lin, A. Pinkus and S. Schocken, *Multilayer feedforward networks with a nonpolynomial activation function can approximate any function*, Neural Networks **6** (1993), pp. 861–867.
URL www.sciencedirect.com/science/article/pii/S0893608005801315

[13] McNaughton, R., *A theorem about infinite-valued sentential logic*, Journal of Symbolic Logic **16** (1951), pp. 1–13.

[14] Mundici, D., *Satisfiability in many-valued sentential logic is NP-complete*, Theoretical Computer Science **52** (1987), pp. 145–153.

[15] Mundici, D., *A constructive proof of McNaughton's theorem in infinite-valued logic*, The Journal of Symbolic Logic **59** (1994), pp. 596–602.

[16] Rudin, W., "Principles of Mathematical Analysis," McGraw-Hill, New York, 1976, 3 edition.

[17] Tarela, J., E. Alonso and M. Martínez, *A representation method for PWL functions oriented to parallel processing*, Mathematical and Computer Modelling **13** (1990), pp. 75 – 83.
URL `www.sciencedirect.com/science/article/pii/089571779090090A`

[18] Tarela, J. and M. Martínez, *Region configurations for realizability of lattice piecewise-linear models*, Mathematical and Computer Modelling **30** (1999), pp. 17–27.

A fresh view of linear logic as a logical framework [1]

Carlos Olarte [2]

ECT, Universidade Federal do Rio Grande do Norte

Elaine Pimentel [3]

DMAT, Universidade Federal do Rio Grande do Norte

Bruno Xavier [4]

DIMAp, Universidade Federal do Rio Grande do Norte

Abstract

One of the most fundamental properties of a proof system is *analyticity*, expressing the fact that a proof of a given formula F only uses subformulas of F. In sequent calculus, this property is usually proved by showing that the cut rule is admissible, *i.e.*, the introduction of the auxiliary lemma A in the reasoning "if A follows from B and C follows from A, then C follows from B" can be eliminated. Mathematically, this means that we can inline the intermediate step A to have a direct proof of C from the hypothesis B. More importantly, the proof of cut-elimination shows that the proof of C follows directly from the axiomatic theory and B (and no external lemmas are needed). The proof of cut-elimination is usually a tedious process through several proof transformations, thus requiring the assistance of (semi-)automatic procedures to avoid mistakes. In a previous work by Miller and Pimentel, linear logic (LL) was used as a logical framework for establishing sufficient conditions for cut-elimination of object logics (OL). The OL's inference rules were encoded as an LL theory and an easy-to-verify criterion sufficed to establish the cut-elimination theorem for the OL at hand. Using such procedure, analyticity of logical systems such as LK (classical logic), LJ (intuitionistic logic) and substructural logics such as MALL (multiplicative additive LL) was proved within the framework. However, there are many logical systems that cannot be adequately encoded in LL, the most symptomatic cases being sequent systems for modal logics. In this paper we use a linear-nested sequent (LNS) presentation of SLL (a variant of linear logic with subexponentials) and show that it is possible to establish a cut-elimination criterion for a larger class of logical systems, including LNS proof systems for K, 4, KT, KD, S4 and the multi-conclusion LNS system for intuitionistic logic (mLJ). Impressively enough, the sufficient conditions for cut-elimination presented here remain as simple as the one proposed by Miller and Pimentel. The key ingredient in our developments is the use of the right formalism: we adopt LNS based OL systems, instead of sequent ones. This not only provides a neat encoding procedure of OLs into SLL, but it also allows for the use of the meta-theory of SLL to establish fundamental meta-properties of the encoded OLs. We thus contribute with procedures for checking cut-elimination of several logical systems that are widely used in philosophy, mathematics and computer science.

Keywords: linear logic, cut elimination

1 Introduction

Apart from formalizing reasoning, proof systems are important tools for analyzing structural properties of proofs, as well as their computational and meta-logical consequences. In particular, one of the main subjects of interest in proof theory is to determine when a proof system supports a notion of *analytic proofs.*

[1] Olarte and Pimentel are funded by CNPq.

[2] Email: carlos.olarte@gmail.com

[3] Email: elaine.pimentel@gmail.com

[4] Email: bruno_xavier86@yahoo.com.br

Analytic calculi consist solely of rules that compose the formulas to be proved in a stepwise manner. As a result, proofs from an analytic calculus satisfy the subformula property: every formula that appears (anywhere) in the proof must be a subformula of the formulas to be proved. This is a powerful restriction on the shape of proofs and can be exploited to prove important meta-logical properties of the logical system such as consistency, decidability and interpolation.

Since analyticity is a highly non-trivial and powerful property, it is natural to ask

(i) how to construct an analytic calculus for a logic of interest; and
(ii) given a pre-existing proof system for a certain logic, how can we determine if it is analytic.

Regarding (i), the best known formalism for proposing analytic proof systems is Gentzen's *sequent calculus* [7]. While its simplicity makes it an ideal tool for proving meta-logical properties, sequent systems are not expressive enough for constructing analytic calculi for many logics of interest. As a result, many new formalisms extending sequent systems have been proposed over the last 30 years, including *hypersequent calculi* [2], *nested calculi* [3, 26] and *labeled calculi* [28]. While such more expressive formalisms enable calculi for a broader class of logics, the greater bureaucracy makes it harder to prove meta-logical properties, such as analyticity itself. Hence the importance of answering (ii).

Since a specific logic gives rise to a specific set of rules in different calculi, it is important to determine whether there is a *general* methodology for determining/analyzing meta-level properties as analyticity. This is the role of logical frameworks in proof theory, where proof systems are adequately embedded into a meta-level formal system so that object-level properties can be uniformly proven. Since logical frameworks often come with automated procedures, the meta-level machinery can be used for proving properties of the embedded systems automatically.

In sequent calculus systems, analyticity is often guaranteed by proving a property called *cut-elimination*, that is, the possibility of eliminating the cut rule below

$$\frac{\Gamma_1 \vdash \Delta_1, A \quad A, \Gamma_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathsf{cut}$$

Intuitively, the cut rule expresses, in logic, the mathematical use of lemmas in proofs: if A follows from B and C follows from A, then C follows from B. That is, one can *cut* the intermediate lemma A. Cut-elimination implies that this deviation through A is not necessary. *Mathematically*, this means that proofs follow directly from the axiomatic theory. *Computationally*, it implies that systems do not have to guess lemmas. *Logically*, cut elimination often implies consistency.

Note that the cut rule has an inherent *duality*: the cut formula A is both a conclusion of a statement and a hypothesis of another. In analytic systems, this duality is often an *invariant*, being preserved throughout the cut-elimination process. Developing general methods for detecting such invariants enables the use of meta-level frameworks to uniformly reasoning about object-level properties.

One of such methods was developed in [17], where *bipoles* and the *focusing proof strategy* [1] in linear logic (LL) [8] were used in order to specify sequent systems and provide sufficient meta-level conditions for cut-elimination. These results were recently formalized in Coq [6]. The main idea is that, by interpreting object-level inference rules as meta-level bipoles, focusing forces a one-to-one correspondence between the application of rules and the derivation of formulas. This completely ties object-level (formulas, rules, derivations) to linear logic, thus showing that if the meta-level use of cuts in represented proofs can be eliminated then all instances of the object-level cut rule can be actually eliminated.

There are, at least, three minimum requirements for the success of logical frameworks to deal with analyticity in object-logics: (a) the spectrum of object-logics amenable for encoding should be representative; (b) an effective and uniform way of deciding the elimination of object-level cuts (based on the meta-level theory) should be stated; and (c) the encoding should be simple (if possible, even automatic). Otherwise, the method might be at risk of (a) not being representative; (b) simply bringing a case-by-case analysis to a more involved framework; or (c) being too complicated to be used or implemented. The main criticism of the work in [17] is (a). The reason for the lack of expressiveness is the fact that LL exponentials can only capture modal behaviors matching exactly their own. The fact that exponentials alone are not enough for separating different types of sequent contexts (sets, multisets or lists of formulas) is also problematic. In order to fix this last issue, in [21] the meta-logic was enhanced to support *subexponentials* (SELL [19]), which can be thought as linear multi-modalities organized as a poset. Intuitively, subexponentials separate the context into locations where encoded formulas are stored, and different order structures would give rise to different hierarchies between them. This for sure enhanced the power of the framework, but at a cost of making both the encoding and the reasoning quite intractable, thus failing (c).

But the meta-level logical system should not be the only one to blame: there are some sequent systems that simply are not adequate themselves for representing a given logic. The case of modal logics is emblematic. Undoubtedly, there are sequent calculi for a number of modal logics exhibiting many good properties (such as analyticity) which can be used in complexity-optimal decision procedures. However, their construction often seems ad-hoc, they are usually not modular, and they mostly lack philosophically relevant properties such as separate left and right introduction rules for the modalities. These problems are often connected to the fact that the modal rules in such calculi usually introduce more than one connective at a time, *e.g.* as in the standard presentation of the rule k for modal logic K:

$$\frac{G_1, \ldots, G_n \vdash F}{\Box G_1, \ldots, \Box G_n \vdash \Box F}\ \mathsf{k}$$

This formulation is somehow dissatisfying since it modifies the context (by adding boxes to the hypothesis) and one loses the distinction between left and right rules for the modal connective box. One way of solving this problem is by considering extensions of the sequent framework that are expressive enough for capturing these modalities using separate left and right introduction rules. One of such extensions is *linear nested systems* (LNS) [11], where a single sequent is replaced with a list of sequents and successors of a sequent (linear nestings) are interpreted under a given modality. The modal rules of these calculi govern the transfer of (modal) formulas between the different sequents, and they are *local*, in the sense that it is sufficient to transfer only one formula at a time.

Interestingly enough, linear nested systems *are* amenable for being encoded in (plain-vanilla) LL [13,14]. The trick is to add labels to nestings and consider labeled formulas in the object-level. This has many interesting consequences, such as the possibility of building uniform linear logic based theorem provers. But still the meta-level characterization of cut-elimination invariants for LNS formalisms remains an open problem, since there is no easy way to reason about them. The bottom line is: there seems to be no simple, perfect solution for this problem.

With this in mind, in this paper we propose a hybrid approach: we will show how to combine the simplicity of LL encodings with the power of subexponentials, but in a different setting and in a very controlled way. We shall built on an LNS system for SLL, a variant of SELL where subexponentials can assume also modal axioms, other than only structural ones (see [12]). Hence the idea is not to change the meta-logic itself, but rather to change the *formalism*: object- and meta-logics are represented as linear nested systems (instead of sequent systems). This actually entails a smooth extension of the work in [17], since all the encodings in that work remain almost the same. And, more impressively, the (natural, simple, efficient) meta-level characterization of object-level cut-elimination in the *op. cit.* also remains unchanged. This provides a neater solution to the problem than the one reported in [21] (using sequent systems as formalism and subexponential LL as meta-logic), where the necessary conditions and the encodings are more involved and not all the modal logics considered here can be handled.

Organization and contributions. We start in Section 2 recalling linear nested systems and how they handle, in a modular way, different modal logical systems. Section 3 builds on [12] and propose an end-active focused system for SLL. Section 4 defines encodings for several (modal) object-logics, some of them not considered in [21] nor in [17]. The translation is natural and the proof of adequacy is immediate due to the focusing discipline. The criteria for establishing cut-elimination for object-logics are presented in Section 5. We show that such criteria can be easily checked. In this section we also present our main result showing how to eliminate the cut rule at the object-level. Section 6 concludes the paper.

2 Linear nested systems

In this section we present a friendly introduction to linear nested systems [11]. The main benefit of this framework is that it exhibits the essential structure to obtain modular calculi, while retaining a very close connection to the ordinary sequent framework [14].

Coming back to the rule k in the introduction, observe that it can also be seen as an infinite set of rules [29]

$$\left\{ \frac{G_1, \ldots, G_n \vdash F}{\Box G_1, \ldots, \Box G_n \vdash \Box F}\ \mathsf{k}_n \mid n \geq 0 \right\}$$

each with a fixed number of principal formulas. This suggests that k could actually be replaced with two rules: one handling the right box and another dealing with the left boxes, one at a time. For being able to do that, we need a tighter control of formulas in the context, something that sequents do not provide. Hence the need

$$\frac{}{\mathcal{G}/\!\!/\Gamma, A \vdash A, \Delta}\ \mathsf{init} \qquad \frac{}{\mathcal{G}/\!\!/\Gamma, \mathtt{f} \vdash \Delta}\ \mathtt{f}_L \qquad \frac{}{\mathcal{G}/\!\!/\Gamma \vdash \mathtt{t}, \Delta}\ \mathtt{t}_R$$

$$\frac{\mathcal{G}/\!\!/\Gamma, F \vdash \Delta \quad \mathcal{G}/\!\!/\Gamma, G \vdash \Delta}{\mathcal{G}/\!\!/\Gamma, F \vee G \vdash \Delta}\ \vee_L \qquad \frac{\mathcal{G}/\!\!/\Gamma \vdash F, \Delta}{\mathcal{G}/\!\!/\Gamma \vdash F \vee G, \Delta}\ \vee_{R1} \qquad \frac{\mathcal{G}/\!\!/\Gamma \vdash G, \Delta}{\mathcal{G}/\!\!/\Gamma \vdash F \vee G, \Delta}\ \vee_{R2} \qquad \frac{\mathcal{G}/\!\!/\Gamma \vdash F, \Delta \quad \mathcal{G}/\!\!/\Gamma \vdash G, \Delta}{\mathcal{G}/\!\!/\Gamma \vdash F \wedge G, \Delta}\ \wedge_R$$

$$\frac{\mathcal{G}/\!\!/\Gamma, F \vdash \Delta}{\mathcal{G}/\!\!/\Gamma, F \wedge G \vdash \Delta}\ \wedge_{L1} \qquad \frac{\mathcal{G}/\!\!/\Gamma, G \vdash \Delta}{\mathcal{G}/\!\!/\Gamma, F \wedge G \vdash \Delta}\ \wedge_{L2} \qquad \frac{\mathcal{G}/\!\!/\Gamma_1 \vdash F, \Delta_1 \quad \mathcal{G}/\!\!/\Gamma_2, G \vdash \Delta_2}{\mathcal{G}/\!\!/\Gamma_1, \Gamma_2, F \to G \vdash \Delta_1, \Delta_2}\ \to_L \qquad \frac{\mathcal{G}/\!\!/\Gamma, F \vdash G, \Delta}{\mathcal{G}/\!\!/\Gamma \vdash F \to G, \Delta}\ \to_R$$

Fig. 1. Propositional rules of the system $\mathsf{LNS_G}$ for classical logic. In the init rule, A is atomic.

$$\frac{\mathcal{G}/\!\!/\Gamma, F, F \vdash \Delta}{\mathcal{G}/\!\!/\Gamma, F \vdash \Delta}\ \mathsf{C}_L \qquad \frac{\mathcal{G}/\!\!/\Gamma \vdash F, F, \Delta}{\mathcal{G}/\!\!/\Gamma \vdash F, \Delta}\ \mathsf{C}_R \qquad \frac{\mathcal{G}/\!\!/\Gamma \vdash \Delta}{\mathcal{G}/\!\!/\Gamma, F \vdash \Delta}\ \mathsf{W}_L \qquad \frac{\mathcal{G}/\!\!/\Gamma \vdash \Delta}{\mathcal{G}/\!\!/\Gamma \vdash F, \Delta}\ \mathsf{W}_R$$

Fig. 2. The structural rules of contraction and weakening.

$$\frac{\Gamma \vdash \Delta /\!\!/ \Sigma, F \vdash \Pi}{\Gamma, F \vdash \Delta /\!\!/ \Sigma \vdash \Pi}\ \mathtt{lift} \qquad \frac{\Gamma \vdash \Delta /\!\!/ F \vdash G}{\mathcal{G}/\!\!/\Gamma \vdash F \supset G, \Delta}\ \supset_R \qquad \frac{\mathcal{G}/\!\!/\Gamma_1 \vdash F, \Delta_1 \quad \mathcal{G}/\!\!/\Gamma_2, G \vdash \Delta_2}{\mathcal{G}/\!\!/\Gamma_1, \Gamma_2, F \supset G \vdash \Delta_1, \Delta_2}\ \supset_L$$

Fig. 3. Some rules of $\mathsf{LNS_I}$ for propositional intuitionistic logic.

for extending the notion of sequent systems.

Definition 2.1 The set LNS of *linear nested sequents* is given recursively by:
(i) if $\Gamma \vdash \Delta$ is a sequent then $\Gamma \vdash \Delta \in \mathsf{LNS}$
(ii) if $\Gamma \vdash \Delta$ is a sequent and $\mathcal{G} \in \mathsf{LNS}$ then $\Gamma \vdash \Delta /\!\!/ \mathcal{G} \in \mathsf{LNS}$.

We call each sequent in a linear nested sequent a *component* and slightly abuse notation, abbreviating "linear nested sequent" to LNS. We shall denote by $\mathsf{LNS}_{\mathcal{L}}$ a linear nested sequent system for a logic $\mathcal{L}$. □

In words, a linear nested sequent is simply a finite list of sequents that matches exactly the *history* of a backwards proof search in an ordinary sequent calculus [11, 14]. We can now adequately represent the local behavior of modalities in the rule k:

$$\frac{\mathcal{G}/\!\!/\Gamma \vdash \Delta /\!\!/ \cdot \vdash F}{\mathcal{G}/\!\!/\Gamma \vdash \Delta, \Box F}\ \Box_R \qquad \frac{\mathcal{G}/\!\!/\Gamma \vdash \Delta /\!\!/ \Gamma', F \vdash \Delta'}{\mathcal{G}/\!\!/\Gamma, \Box F \vdash \Delta /\!\!/ \Gamma' \vdash \Delta'}\ \Box_L$$

Reading bottom up, while in $\Box_R$ a new nesting/component is created and F is moved there, in $\Box_L$ *exactly one* boxed formula moves into an existing nesting, losing its modality.

We will explore the local/linear structure of LNS in two ways: first, components have a tight connection to *worlds* in Kripke-like semantics, so that LNS is an adequate framework for describing the behavior of alethic modalities in certain logical systems; and second, on fragmenting information into components, rules act locally on formulas, hence often being context independent, so that the movement of formulas on derivations can be better predicted and controlled. This implies that both: we will be able to adequately specify a representative class of logical systems; and many techniques developed in [17] will remain valid in the proposed framework.

A further advantage of this framework is that it is often possible to restrict the list of sequents in a LNS to the last 2 components, that we call *active.*

Definition 2.2 An application of a linear nested sequent rule is *end-active* if the rightmost components of the premises are active and the only active components (in premise and conclusion) are the two rightmost ones. The *end-active variant* of a LNS calculus is the calculus with the rules restricted to end-active applications.

All the logical systems studied in [11, 12, 14] can be restricted to the end-active version. Figs. 1 and 2 present the end-active $\mathsf{LNS_G}$ for the classical propositional connectives and the structural rules of weakening and contraction. Observe that, when restricted to classical logic, new components are never created (this reflects the fact that the Kripke structure for classical logic is flat). Hence the LNS collapses to the usual sequent system LK [7]. A more interesting case is the linear nested system for propositional intuitionistic logic. $\mathsf{LNS_I}$ [11] is the system sharing with $\mathsf{LNS_G}$ the axioms, structural rules and rules for conjunction and disjunction, but adding the rules for intuitionistic implication $\supset$ shown in Fig. 3. Observe that, bottom-up, the rule for implication right creates a new component, adds the sequent $F \vdash G$ there and erases the back history.

Axioms: K $\Box(F \supset G) \supset (\Box F \supset \Box G)$ D $\neg(\Box F \wedge \Box\neg F)$ T $\Box F \supset F$ 4 $\Box F \supset \Box\Box F$

$$\frac{\Gamma \vdash \Delta /\!\!/ \Sigma, F \vdash \Pi}{\Gamma, \Box F \vdash \Delta /\!\!/ \Sigma \vdash \Pi}\ \Box_L \qquad \frac{\Gamma \vdash \Delta /\!\!/ \cdot \vdash F}{\mathcal{G} /\!\!/ \Gamma \vdash \Delta, \Box F}\ \Box_R \qquad \frac{\Gamma \vdash \Delta /\!\!/ F \vdash \cdot}{\mathcal{G} /\!\!/ \Gamma, \Box F \vdash \Delta}\ \mathsf{d} \qquad \frac{\mathcal{G} /\!\!/ \Gamma, F \vdash \Delta}{\mathcal{G} /\!\!/ \Gamma, \Box F \vdash \Delta}\ \mathsf{t} \qquad \frac{\Gamma \vdash \Delta /\!\!/ \Sigma, \Box F \vdash \Pi}{\Gamma, \Box F \vdash \Delta /\!\!/ \Sigma \vdash \Pi}\ 4$$

Fig. 4. Some modal axioms and their linear nested sequent rules.

The lift rule, on the other hand, moves left formulas into the next component. The consecutive application of these rules mimics, possibly in many steps, the behavior of the sequent right rule for implication in the multi-conclusion intuitionistic sequent system mLJ [16]

$$\dfrac{\Gamma, F \vdash G}{\Gamma \vdash \Delta, F \supset G}\ \supset_R \atop \vdots\ \mathcal{G} \qquad \rightsquigarrow \qquad \dfrac{\dfrac{\cdot \vdash \Delta /\!\!/ \Gamma, F \vdash G}{\Gamma \vdash \Delta \ /\!\!/ F \vdash G}\ \texttt{lift}}{\mathcal{G} /\!\!/ \Gamma \vdash \Delta, F \supset G}\ \supset_R$$

This also interprets, proof theoretically, the definition of satisfaction for intuitionistic logic (see [24] for more details). Observe that, once all formulas in the left context are lifted, the only possible action is the application of rules in the last (right-most) component. Hence the right context Δ is *forgotten*. This shows an interesting dynamic in end-active systems: apply first rules that do not involve *moving-between* or *creating-new* components. After creating new components, apply the lift-kind rules as much as possible. Then forget about past components and move forward, reasoning over the new components.

The possibility of having such a notion of "proof normalization" was studied in [25] in the nested systems framework. In that work, it was shown that end-active nested systems with very specific rules' shape can be *sequentialized*. This implies that such nested systems correspond to well known sequent systems. In this work, we will use this result in a very pragmatic way. Namely, since some sequent systems are not adequate for specification and reasoning, we will consider the corresponding (end-active) LNS that: have the same meta-logical properties; can be easily specified; and entails easy meta-level conditions for cut-elimination.

In the present work, besides reasoning about intuitionistic and classical logics, we shall also reason about linear nested systems for some notable extensions of the normal modal logic K. Fig. 4 presents some modal axioms and the respective linear nested rules. The calculus $\mathsf{LNS_K}$ contains the rules of $\mathsf{LNS_G}$ together with the rules $\Box_R$ and $\Box_L$. Extensions of the logic K are represented by KR, where R is the list of the respective axioms. As usual, we write S4 = KT4.

3 Linear logic and its variants

Linear logic (LL) [8] 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 !). Formulas marked with ? behave *classically*, i.e., they can be contracted (duplicated) and weakened (erased) during proofs. LL connectives include the additive conjunction & and disjunction $\oplus$ and their multiplicative versions $\otimes$ and $⅋$, together with their units and the first-order quantifiers:

$$\begin{array}{rcccccccccccccc} F, G, \ldots & ::= & A & \mid & F \otimes G & \mid & 1 & \mid & F \oplus G & \mid & 0 & \mid & \exists x.F & \mid & !F \\ & \mid & A^{\perp} & \mid & F ⅋ G & \mid & \perp & \mid & F \& G & \mid & \top & \mid & \forall x.F & \mid & ?F \\ & & \text{LITERALS} & & \multicolumn{3}{c}{\text{MULTIPLICATIVES}} & & \multicolumn{3}{c}{\text{ADDITIVES}} & & \text{QUANTIF.} & & \text{EXP.} \end{array}$$

Note that $(\cdot)^{\perp}$ (negation) has atomic scope. For an arbitrary formula F, $F^{\perp}$ denotes the result of moving negation inward until it has atomic scope. We shall refer to atomic (A) and negated atomic ($A^{\perp}$) formulas as literals. The connectives in the first line denote the de Morgan dual of the connectives in the second line. Hence, for atoms A, B, the expression $(\perp \& (A \otimes (!B)))^{\perp}$ denotes $1 \oplus (A^{\perp} ⅋ (?B^{\perp}))$. The linear implication $F \multimap G$ is a short hand for $F^{\perp} ⅋ G$. The equivalence $F \equiv G$ is defined as $(F \multimap G) \& (G \multimap F)$.

The usual rules for the exponentials in LL in its one-sided sequent presentation are

$$\frac{\vdash ?G_1, \cdots, ?G_n, F}{\vdash ?G_1, \cdots, ?G_n, !F}\ ! \qquad \frac{\vdash \Gamma, F}{\vdash \Gamma, ?F}\ ? \qquad \frac{\vdash \Gamma}{\vdash \Gamma, ?F}\ ?_W \qquad \frac{\vdash \Gamma, ?F, ?F}{\vdash \Gamma, ?F}\ ?_C$$

Note that, in order to introduce ! (this rule is usually called *promotion*) all the formulas must be marked with ?. Clearly, this rule is not context-independent (compare it with the rule k in the introduction). The other three rules correspond to dereliction, weakening and contraction.

In [9, 27], systems with partially local rules for LL were proposed. In [12], the end-active $\mathsf{LNS_{LL}}$ system for

Negative rules:

$$\frac{}{\vdash \Theta;\Gamma \Uparrow \top, L}\ \top \qquad \frac{\vdash \Theta;\Gamma \Uparrow L}{\vdash \Theta;\Gamma \Uparrow \bot, L}\ \bot \qquad \frac{\vdash \Theta;\Gamma \Uparrow F, G, L}{\vdash \Theta;\Gamma \Uparrow F \parr G, L}\ \parr \qquad \frac{\vdash \Theta;\Gamma, S \Uparrow L}{\vdash \Theta;\Gamma \Uparrow S, L}\ \mathsf{store}$$

$$\frac{\vdash \Theta;\Gamma \Uparrow F, L \quad \vdash \Theta;\Gamma \Uparrow G, L}{\vdash \Theta;\Gamma \Uparrow F \,\&\, G, L}\ \& \qquad \frac{\vdash \Theta;\Gamma \Uparrow F[y/x], L}{\vdash \Theta;\Gamma \Uparrow \forall x.F, L}\ \forall \qquad \frac{\vdash \Theta, i : F;\Gamma \Uparrow L}{\vdash \Theta;\Gamma \Uparrow ?^i F, L}\ \mathsf{store}_c$$

Positive rules:

$$\frac{\vdash \Theta;\Gamma_1 \Downarrow F \quad \vdash \Theta;\Gamma_2 \Downarrow G}{\vdash \Theta;\Gamma_1,\Gamma_2 \Downarrow F \otimes G}\ \otimes \qquad \frac{\vdash \Theta;\Gamma \Downarrow F_i}{\vdash \Theta;\Gamma \Downarrow F_1 \oplus F_2}\ \oplus_i \qquad \frac{\vdash \Theta;\Gamma \Downarrow F[t/x]}{\vdash \Theta;\Gamma \Downarrow \exists x.F}\ \exists \qquad \frac{}{\vdash \Theta;\cdot \Downarrow 1}\ 1$$

Id, Decide and Release:

$$\frac{}{\vdash \Theta; A \Downarrow A^{\perp}}\ \mathsf{I_l} \qquad \frac{}{\vdash \Theta, i : A;\cdot \Downarrow A^{\perp}}\ \mathsf{I_c} \qquad \frac{\vdash \Theta;\Gamma \Downarrow P}{\vdash \Theta;\Gamma, P \Uparrow \cdot}\ \mathsf{D_l} \qquad \frac{\vdash \Theta, i : P_a;\Gamma \Downarrow P_a}{\vdash \Theta, i : P_a;\Gamma \Uparrow \cdot}\ \mathsf{D_c} \qquad \frac{\vdash \Theta;\Gamma \Uparrow N}{\vdash \Theta;\Gamma \Downarrow N}\ \mathsf{R_n}$$

Subexponentials:

$$\frac{\vdash \Theta, j : F;\cdot \Uparrow \cdot /\!\!/^i \vdash \Upsilon;\cdot \Uparrow L, F}{\vdash \Theta, j : F;\cdot \Uparrow \cdot /\!\!/^i \vdash \Upsilon;\cdot \Uparrow L}\ ?^i{}_{\mathsf{k}}\ (\text{for } i \preceq j) \qquad \frac{\vdash \Theta, j : F;\Gamma \Uparrow \cdot /\!\!/^i \vdash \Upsilon, j : F;\cdot \Uparrow L}{\vdash \Theta, j : F;\Gamma \Uparrow \cdot /\!\!/^i \vdash \Upsilon;\cdot \Uparrow L}\ ?^i{}_{\mathsf{4}}\ (\text{for } i \preceq j \text{ and } \mathsf{4} \in \mathcal{U}(j))$$

$$\frac{\vdash \Theta, i : F;\cdot \Uparrow \cdot /\!\!/^i \vdash \cdot;\cdot \Uparrow F}{\vdash \Theta, i : F;\cdot \Uparrow \cdot}\ \mathsf{D_d}\ (\text{for } \mathsf{D} \in \mathcal{U}(i)) \qquad \frac{\vdash \Theta;\cdot \Uparrow \cdot /\!\!/^i \vdash \cdot;\cdot \Uparrow F}{\vdash \Theta;\cdot \Downarrow\, !^i F}\ !^i \qquad \frac{\vdash \Theta;\cdot \Uparrow L}{\vdash \Upsilon;\cdot \Uparrow \cdot /\!\!/^i \vdash \Theta;\cdot \Uparrow L}\ \mathsf{R_r}$$

Fig. 5. End-active focused system $\mathsf{LNS_{FSLL}}$. In $\mathsf{I_c}$ and $\mathsf{I_l}$, A is an atom. In $\forall$, y is fresh. In store, S is a literal or a positive formula. In $\mathsf{D_c}$, P_a is not atomic and in $\mathsf{D_l}$, P is a positive formula. In $\mathsf{R_n}$, N is a negative formula. In $\mathsf{D_c}$ and $\mathsf{I_c}$, $\mathsf{T} \in \mathcal{U}(i)$.

LL was introduced, making the rule for ! local, in the sense that one does not need to check the sequent context in order to apply promotion:

$$\frac{\vdash \Gamma /\!\!/ \vdash F}{\mathcal{E} /\!\!/ \vdash \Gamma, !F}\ ! \qquad \frac{\vdash \Gamma /\!\!/ \vdash \Delta, ?F}{\vdash \Gamma, ?F /\!\!/ \vdash \Delta}\ ?$$

where $\mathcal{E}$ is an empty list of components. Note the similarities between the LNS rules ! and $\Box_R$; and ? and 4 in Fig. 4. Indeed, the work [12] exploits such similarities to propose extensions of $\mathsf{LNS_{LL}}$ with *subexponentials* where the exponentials are decorated with labels allowing for different modal behaviors.

Since the proof of adequacy of the proposed encodings in Sec. 4 is greatly alleviated if a *focusing discipline* is used, we introduce next the focused version of the LNS for linear logic with subexponentials.

3.1 Multi-modalities in linear logic and the focused system $\mathsf{LNS_{FSLL}}$

As exponentials $(!, ?)$ in linear logic can be seen, roughly, as modalities in modal logic, subexponentials are nothing else than *multi-modalities*. Intuitively, this means that we can mark the exponentials with *labels* taken from a set $\mathcal{S}$ organized in a pre-order $\preceq$, obtaining (possibly infinitely-many) exponentials ($!^i, ?^i$ for $i \in \mathcal{S}$). Also like modal connectives, subexponentials are not *canonical* [4], in the sense that if $i \neq j$ then $!^i F \not\equiv\, !^j F$ and $?^i F \not\equiv ?^j F$. Moreover, the pre-order determines the provability relation: $!^b F$ *implies* $!^a F$ iff $a \preceq b$.

The main difference between multi-modalities and subexponentials is that the last carries the possibility of having different structural behaviors, being *unbounded* (or classical) if weakening and contraction are allowed or *bounded* otherwise (thus having a linear behavior).

This opened a venue for proposing different multi-modal substructural logical systems, that encountered a number of different applications *e.g.* in the specification and verification of concurrent systems [20], biological systems [22], applications in linguistics [10], and the specification of systems with multiple contexts, which may be represented by sets or multisets of formulas [21].

In [12] we extended the concept of *simply dependent multimodal logics* [5] (SDML) to the substructural case, where subexponentials considered not only the structural axioms for contraction and weakening, but also axioms for modalities $\{\mathsf{K}, \mathsf{4}, \mathsf{D}, \mathsf{T}\}$ (see Fig. 4) for the subexponentials. This means that $?^i$ can behave classically or not, but also with exponential behaviors different from those in LL. Hence, by assigning different modal axioms one obtains, in a modular way, a class of different substructural modal logics. For instance, subexponentials assuming T allow for dereliction, those assuming 4 are persistent (while those assuming only K are not) and D forbids both persistency and dereliction – in fact, substructural KD can be seen as a fragment of light linear logic LLL [9].

We consider here only classical versions of SDML, that we call SLL, where subexponentials are unbounded. Although this is not a necessary restriction, it is enough for specifying the classical based systems considered in this paper, and it simplifies the notation of the resulting system.

Definition 3.1 The SLL subexponential signature is given by $\Sigma = \langle \mathcal{S}, \preceq, \mathcal{U}(i)\rangle$, where $\mathcal{S}$ is a set of unbounded labels, $\mathcal{U}(i)$ represent the set of axioms within $\{\mathsf{K}, \mathsf{4}, \mathsf{D}, \mathsf{T}\}$ that the subexponential $?^i \in \mathcal{S}$ assumes, and $\preceq$ is a pre-order among the elements of $\mathcal{S}$ that is upwardly closed with respect to $\mathcal{U}(i)$, *i.e.*, if $\mathsf{A} \in \mathcal{U}(i)$ and $i \preceq j$, then $\mathsf{A} \in \mathcal{U}(j)$.

The proof system for SLL is constructed by adding all the rules for the linear logic connectives except for the exponentials. The rules for subexponentials are added according to the subexponential signature Σ. We will present next the focused variant of the linear nested system for SLL.

Focusing [1] is a discipline on proofs aiming at reducing non-determinism during proof search. Focusing in LL based systems is grounded on two kinds of separations: (i) classical/linear behaving formulas and (ii) invertible/non invertible introduction rules.

For, (i), observe that it is possible to incorporate the structural rules of contraction and weakening for formulas of the shape $?F$ into the LL introduction rules. This is reflected into the syntax in the so called *dyadic sequents* where the context is split into two: a classical (set of formulas Θ) and a linear (multiset of formulas Γ). The dyadic sequent $\vdash \Theta : \Gamma$ is then interpreted as the linear logic sequent $\vdash ?\Theta, \Gamma$ where $?\Theta = \{?F \mid F \in \Theta\}$. This can be easily generalized to the case of the subexponentials: the classical context is a partition $\Theta = \{i : \Theta_i \mid i \in \mathcal{S}\}$, and the dyadic sequent $\vdash (i : \Theta_i)_{i \in \mathcal{S}} : \Gamma$ is interpreted as the (subexponential) linear logic sequent $\vdash (?^i\Theta_i)_{i\in\mathcal{S}}, \Gamma$.

For (ii), it turns out that proofs can be organized in two alternating phases: the negative phase containing only invertible rules, and the positive phase contains only non-invertible rules. The connectives $\parr, \bot, \&, \top, ?^i, \forall$ have invertible introduction rules and are thus classified as *negative*. The remaining connectives $\otimes, 1, \oplus, 0, !^i, \exists$ are *positive*. Formulas inherit their polarity according to their main connective, *e.g.*, $F \otimes G$ is positive and $F \parr G$ is negative. Although the bias assigned to atoms does not interfere with provability [18], here we follow Andreoli's convention of classifying atomic formulas as *negative*, thus negated atoms as positive.

In the focused system $\mathsf{LNS}_{\mathsf{FSLL}}$ (Fig. 5), dyadic (linear nested) sequents are further refined, so to reflect not only the negative/positive proof phases described above, but also the behavior of the promotion rule:

- $\vdash \Theta; \Gamma \Uparrow L$ belongs to the *negative phase*. During this phase, all negative formulas in the list L are introduced and all positive formulas and literals are moved to the linear context Γ.

- $\vdash \Theta; \Gamma \Downarrow F$ belongs to the *positive phase*, where all positive connectives at the root of F are introduced.

- $\mathcal{G} /\!\!/^i \vdash \Theta; \Gamma \Uparrow F$ belongs to the *exponential phase*. During this phase, only applications of the rules for $?^i$ are allowed, ending with an application of $\mathsf{R_r}$.

Reading the rules bottom-up, the ones belonging to the negative phase pick the first formula F on the list L. Negative formulas are eagerly decomposed, while literals and positive formulas are stored into the linear context, as shown in the left derivation below.

$$\dfrac{\dfrac{\dfrac{\vdash \Theta, i : B; \Gamma, F \otimes G, A^{\bot} \Uparrow L}{\vdash \Theta; \Gamma, F \otimes G, A^{\bot} \Uparrow ?^iB, L}\ \mathsf{store}_c}{\vdash \Theta; \Gamma \Uparrow F \otimes G, A^{\bot}, ?^iB, L}\ 2 \times \mathsf{store}}{\vdash \Theta; \Gamma \Uparrow (F \otimes G) \parr A^{\bot}, ?^iB, L}\ \parr \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash \cdot; A, F \oplus !^iA \Uparrow \cdot}{\vdash \cdot; \cdot \Uparrow A, F \oplus !^iA}\ 2 \times \mathsf{store}}{\vdash \Theta, j : F \oplus !^iA; \cdot \Uparrow \cdot /\!\!/^i \vdash \cdot; \cdot \Uparrow A, F \oplus !^iA}\ \mathsf{R_r}}{\vdash \Theta, j : F \oplus !^iA; \cdot \Uparrow \cdot /\!\!/^i \vdash \cdot; \cdot \Uparrow A}\ ?^i{}_{\mathsf{k}}}{\vdash \Theta, j : F \oplus !^iA; \cdot \Downarrow !^iA}\ !^i}{\vdash \Theta, j : F \oplus !^iA; \cdot \Downarrow F \oplus !^iA}\ \oplus_2}{\vdash \Theta, j : F \oplus !^iA; \cdot \Uparrow \cdot}\ \mathsf{D_c} \tag{1}$$

The negative phase ends when the list L is empty. Then the proof moves to an exponential phase by the application of the rule $\mathsf{D_d}$, or a positive phase by *focusing* on a formula F via the deciding rules $\mathsf{D_l}$ and $\mathsf{D_c}$ (note that F can never be atomic). In $\mathsf{D_l}$, F should be a positive formula taken from the linear context (and thus erased from it). In $\mathsf{D_c}$, a copy of F is taken from the classical context, thus making an implicit contraction *and* a dereliction. Since we are considering only unbounded subexponentials, contraction is not a problem. However, in order to derelict $?^iF$, it should be the case that $\mathsf{T} \in \mathcal{U}(i)$ – this is the side condition in the caption of Fig. 5. Once we focus on a formula, the proof follows by applying positive rules, where the focus persists on the decomposed subformulas until either: a negative formula is reached (and the positive phase ends with the application of $\mathsf{R_n}$); or a banged formula is derived, which creates a new component and triggers an exponential phase execution. At this point, only the rules for $?^i$, moving formulas between components, are allowed. When this moving is over, the exponential phase (and the positive phase) ends with an application of the rule $\mathsf{R_r}$, starting again a negative phase. See the right derivation in Equation (1), where we assume that A is atomic,

$\mathsf{T} \in \mathcal{U}(j)$ and $i \preceq j$.

The proof ends with applications of the initial axioms at the leaves. For an atom A, the proof of $\vdash \Theta; \Gamma \Downarrow A^{\perp}$ must finish immediately with the rule $\mathsf{I_c}$ (and the atomic proposition A must be in a context i s.t. $\mathsf{T} \in \mathcal{U}(i)$) or the rule $\mathsf{I_l}$ (and the linear context is the singleton $\{A\}$). This behavior will be fundamental to understanding the specifications described later. Note the implicit weakening of the (classical) context Θ on the leaves.

In our encodings, we shall consider the following set of labels: $\mathcal{S} = \{l, c, k, t, d, 4, td, t4, d4\}$. We assume that $\mathcal{U}(l) = \{\mathsf{T}\}$, $\mathsf{K} \in \mathcal{U}(i)$ for all $i \neq l$ (hence the rule $?^{i}{}_{\mathsf{k}}$ can be applied for *all* subexponentials *but* l), $\mathcal{U}(c) = \{\mathsf{K}, \mathsf{T}, 4, \mathsf{D}\}$, and all the other labels assume the axioms represented by the same letter, *e.g.* $\mathcal{U}(t4) = \{\mathsf{K}, \mathsf{T}, 4\}$. Hence, for instance, $!^{t4}$ has the same behavior as the box modality in the modal logic $\mathsf{S4}$. Not surprisingly, but interesting enough, $t4$ will also be used for the specification of the intuitionistic implication.

Finally, we will set the following subexponential order $\preceq$ for $\mathcal{S}$: l is not related to any other label, and $i \preceq j$ iff $\mathcal{U}(i) \subseteq \mathcal{U}(j)$. The idea is that l will be the *local* subexponential, that will allow for weakening and contraction *within a component*, that is, in a sequent only. The other subexponentials in $\mathcal{S}$ allow moving information *between components*, that is, in different sequents. This flow of information is then regulated using the order, where greater subexponentials can move formulas to smaller ones.

The following derivation shows that the axiom $\mathsf{4}$ is provable using the subexponential 4.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \cdot; A \Downarrow A^{\perp}}\,\mathsf{I_l}}{\vdash \cdot; A^{\perp}, A \Uparrow \cdot}\,\mathsf{D_l}}{\vdash \cdot;\cdot \Uparrow \cdot /\!/^{4} \vdash \cdot;\cdot \Uparrow A^{\perp}, A}\,\mathsf{R_r}, \mathsf{store}}{\vdash 4 : A^{\perp};\cdot \Uparrow !^{4}A}\,\mathsf{store}, \mathsf{D_l}, !^{4}, ?^{4}{}_{\mathsf{k}}}{\vdash 4 : A^{\perp};\cdot \Uparrow \cdot /\!/^{4} \vdash 4 : A^{\perp};\cdot \Uparrow !^{4}A}\,\mathsf{R_r}}{\vdash 4 : A^{\perp};\cdot \Uparrow \cdot /\!/^{4} \vdash \cdot;\cdot \Uparrow !^{4}A}\,?^{4}{}_{4}}{\vdash 4 : A^{\perp};\cdot \Downarrow !^{4}!^{4}A}\,!^{4}}{\vdash 4 : A^{\perp}; !^{4}!^{4}A \Uparrow \cdot}\,\mathsf{D_l}}{\vdash \cdot;\cdot \Uparrow ?^{4}A^{\perp}, !^{4}!^{4}A}\,\mathsf{store}_{c}, \mathsf{store}}{\vdash \cdot;\cdot \Uparrow ?^{4}A^{\perp} \parr !^{4}!^{4}A}\,\parr
$$

The proof involves three components: the initial sequent and the other two created using the $!^{4}$ rule. The information $A^{\perp}$ is first passed from the 1st to the 2nd components through the classical contexts via rule $?^{4}{}_{4}$, then to the linear context into the 3rd component via rule $?^{4}{}_{\mathsf{k}}$. All the other axioms in Fig. 4 are proven similarly, using the correspondent subexponential.

4 Specifying LNS

In this section we shall encode the logical rules of LNS systems as SLL theories. We shall also prove that such specification is *adequate* in the sense that an object logic (OL) sequent S is provable iff the encoding of S together with the resulting theory of the OL's rules is also provable in $\mathsf{LNS_{SFLL}}$. See [17, 19] for a further discussion about the level of adequacy that can be achieved with this kind of LL specifications.

In [17], LL was used as a logical framework for specifying a number of logical systems. Here we shall proceed similarly but building on SLL. The idea is to use two predicates $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ for identifying objects that appear on the left or on the right side, respectively, of the sequents in the OL. 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 of atomic SLL formulas $\lfloor B_1 \rfloor, \ldots, \lfloor B_n \rfloor, \lceil C_1 \rceil, \ldots, \lceil C_m \rceil$. As a mnemonic, formulas on the (**L**)eft side of object-level sequents are encoded with the predicate starting with $\lfloor$. In the following, given a set of OL formulas Γ, we shall use $\lfloor \Gamma \rfloor$ to denote the set of SLL formulas $\{\lfloor F \rfloor \mid F \in \Gamma\}$. Similarly for $\lceil \Gamma \rceil$.

Inference rules of the OL are specified as rewriting clauses that replace the active formula in the conclusion of the rule by the resulting formulas in the premises. The linear logic connectives indicate how these object-level formulas are connected: contexts are copied ($\&$) or split ($\otimes$), in different inference rules ($\oplus$) or in the same sequent ($\parr$). Such specification clauses will be members of a theory $\mathcal{T}_{\mathcal{L}}$ of the specified rules in SLL of the logical system $\mathcal{L}$. Theories will be stored with the subexponential c. Note that $i \preceq c$ for any $i \in \mathcal{S}$. Hence, the formulas in $\mathcal{T}_{\mathcal{L}}$ can be copied/moved to any component.

As an example, consider the rules $\wedge_{L1}$, $\wedge_{L2}$ and $\wedge_{R}$ for conjunction in Fig. 1. Following the method shown

$$\begin{array}{lll}
\wedge_L : \lfloor A \wedge B\rfloor^\perp \otimes (\lfloor A\rfloor \oplus \lfloor B\rfloor) & \wedge_R : \lceil A \wedge B\rceil^\perp \otimes (\lceil A\rceil \,\&\, \lceil B\rceil) & \mathtt{f}_L : \lfloor \mathtt{f}\rfloor^\perp \otimes \top \\
\vee_L : \lfloor A \vee B\rfloor^\perp \otimes (\lceil A\rceil \,\&\, \lfloor B\rfloor) & \vee_R : \lceil A \vee B\rceil^\perp \otimes (\lceil A\rceil \oplus \lceil B\rceil) & \mathtt{t}_R : \lceil \mathtt{t}\rceil^\perp \otimes \top \\
\rightarrow_L : \lfloor A \rightarrow B\rfloor^\perp \otimes (\lceil A\rceil \otimes \lfloor B\rfloor) & \rightarrow_R : \lceil A \rightarrow B\rceil^\perp \otimes (\lfloor A\rfloor \parr \lceil B\rceil) & \mathsf{init} : \lfloor A\rfloor^\perp \otimes \lceil A\rceil^\perp
\end{array}$$

Fig. 6. Encoding of propositional rules of the system $\mathsf{LNS_G}$ for classical logic. In all the specification clauses, there is an implicit existential quantification on A and B.

$$\mathsf{pos}_i : \lfloor A\rfloor^\perp \otimes (?^i \lfloor A\rfloor) \qquad \mathsf{neg}_i : \lceil A\rceil^\perp \otimes (?^i \lceil A\rceil)$$

Fig. 7. Encoding of the structural rules.

in [17], these rules yield the following SLL clauses (present in Fig. 6)

$$\wedge_L : \exists F, G.(\lfloor F \wedge G\rfloor^\perp \otimes (\lfloor F\rfloor \oplus \lfloor G\rfloor)) \qquad \wedge_R : \exists F, G.(\lceil F \wedge G\rceil^\perp \otimes (\lceil F\rceil \,\&\, \lceil G\rceil))$$

If we decide to focus on the clause $\wedge_L$ from the theory $\mathcal{T}_\mathsf{G}$, there is only one possible course of action, where $\mathsf{I} = \mathsf{I_l}$ or $\mathsf{I} = \mathsf{I_c}$, accordingly

$$\dfrac{\dfrac{\dfrac{}{\vdash \Theta, c : \mathcal{T}_\mathsf{G}; , \Gamma_1 \Downarrow \lfloor F \wedge G\rfloor^\perp}\,\mathsf{I} \quad \dfrac{\dfrac{\vdash \Theta, c : \mathcal{T}_\mathsf{G}; \Gamma_2, \lfloor F\rfloor \Uparrow \cdot}{\vdash \Theta, c : \mathcal{T}_\mathsf{G}; \Gamma_2 \Downarrow \lfloor F\rfloor}\,\mathsf{R_n}, \mathsf{store}}{\vdash \Theta, c : \mathcal{T}_\mathsf{G}; \Gamma_2 \Downarrow \lfloor F\rfloor \oplus \lfloor G\rfloor}\,\oplus_1}{\vdash \Theta, c : \mathcal{T}_\mathsf{G}; \Gamma_1, \Gamma_2 \Downarrow \exists F, G.(\lfloor F \wedge G\rfloor^\perp \otimes (\lfloor F\rfloor \oplus \lfloor G\rfloor))}\,\exists, \exists, \otimes}{\vdash \Theta, c : \mathcal{T}_\mathsf{G}; \Gamma_1, \Gamma_2 \Uparrow \cdot}\,\mathsf{D_c}$$

with $\Gamma_1 = \lfloor F \wedge G\rfloor$, or $\Gamma_1 = \emptyset$ and $\lfloor F \wedge G\rfloor \in \Theta$. Bottom-up, the active formula $F \wedge G$ is taken from the linear or the classical context and the whole positive phase (after the resulting negative phase) ends by storing the atom $\lfloor F\rfloor$ into the linear context. This derivation mimics *exactly* an application of the rule $\wedge_{L1}$ at the object level. Similarly, if instead of $\oplus_1$ we apply $\oplus_2$, the atom $\lfloor G\rfloor$ is stored, thus reflecting the behavior of $\wedge_{L2}$.

If we do the same exercise with $\wedge_R$, the derivation ends up with two premises corresponding exactly to the two premises of the rule $\wedge_R$

$$\dfrac{\dfrac{\dfrac{}{\vdash \Theta, c : \mathcal{T}_\mathsf{G}; \Gamma_1 \Downarrow \lceil F \wedge G\rceil^\perp}\,\mathsf{I} \quad \dfrac{\vdash \Theta, c : \mathcal{T}_\mathsf{G}; \Gamma_2, \lceil F\rceil \Uparrow \cdot \quad \vdash \Theta, c : \mathcal{T}_\mathsf{G}; \Gamma_2, \lceil G\rceil \Uparrow \cdot}{\vdash \Theta, c : \mathcal{T}_\mathsf{G}; \Gamma_2 \Downarrow (\lceil F\rceil \,\&\, \lceil G\rceil)}\,\mathsf{R_n}, \&, \mathsf{store}}{\vdash \Theta, c : \mathcal{T}_\mathsf{G}; \Gamma_1, \Gamma_2 \Downarrow \exists F, G.(\lceil F \wedge G\rceil^\perp \otimes (\lceil F\rceil \,\&\, \lceil G\rceil))}\,\exists, \exists\otimes}{\vdash \Theta, c : \mathcal{T}_\mathsf{G}; \Gamma_1, \Gamma_2 \Uparrow \cdot}\,\mathsf{D_c}$$

Moreover, focusing on the initial clause (see Figure 6) implies finishing the proof (by showing that F is on the left and on the right of the OL sequent)

$$\dfrac{\dfrac{\dfrac{}{\vdash \Theta, c : \mathcal{T}_\mathsf{G}; \Gamma_1 \Downarrow \lceil F\rceil^\perp}\,\mathsf{I} \quad \dfrac{}{\vdash \Theta, c : \mathcal{T}_\mathsf{G}; \Gamma_2 \Downarrow \lfloor F\rfloor^\perp}\,\mathsf{I}}{\vdash \Theta, c : \mathcal{T}_\mathsf{G}; \Gamma_1, \Gamma_2 \Downarrow \exists F. \lfloor F\rfloor^\perp \otimes \lceil F\rceil^\perp}\,\exists, \otimes}{\vdash \Theta, c : \mathcal{T}_\mathsf{G}; \Gamma_1, \Gamma_2 \Uparrow \cdot}\,\mathsf{D_c}$$

Regarding the structural rules of weakening and contraction, it may be the case that an OL admits some of them on the left, right or both sides of the sequent. We mimic those behaviors by adding the structural rules in Fig. 7 according to each case. For instance, if pos_i (weakening/contraction for the left context of the OL sequent) is in $\mathcal{T}_\mathcal{L}$, we can prove the equivalence $\lfloor F\rfloor \equiv ?^i \lfloor F\rfloor$. Hence, under the presence of pos_i, we are free to do contraction on atoms of the form $\lfloor F\rfloor$. Similarly for neg_i and right formulas. Observe that pos_i and neg_i are parametric w.r.t. the subexponential label. Hence, for example, if $i = l$, then the use of such clauses is *restricted* to a component (recall that $l \in \mathcal{S}$ is meant to be the *local*, one component, subexponential).

The adequacy of the specification of $\mathsf{LNS_G}$ is the same as showed in [17], since this LNS system coincides

Intuitionistic implication:	$\supset_L$: $\lfloor A \supset B\rfloor^{\perp} \otimes (\lceil A\rceil \otimes \lfloor B\rfloor)$	$\supset_R$: $\lceil A \supset B\rceil^{\perp} \otimes !^{t4}(\lfloor A\rfloor \parr \lceil B\rceil)$
Modal rules:	$\Box_{Li} : \lfloor \Box A\rfloor^{\perp} \otimes ?^{i}\lfloor A\rfloor$	$\Box_{Ri} : \lceil \Box A\rceil^{\perp} \otimes !^{i}\lceil A\rceil$

Fig. 8. Encoding of intuitionistic implication rules and modal rules.

with the usual sequent system LK.

Theorem 4.1 *Let $\mathcal{T}_{\mathsf{G}}$ consist of the set of the specification clauses in Figs. 6 together with the structural rules* pos_i *and* neg_i *where* $\mathsf{T} \in \mathcal{U}(i)$. *Then $\mathcal{T}_{\mathsf{G}}$ is adequate w.r.t.* $\mathsf{LNS_G}$.

Let us move our attention to the intuitionistic case. In [30] we mechanized such an adequacy result [17] for the specification of the sequent system LJ of propositional intuitionistic logic [7], and in [21] we considered the multi-conclusion intuitionistic system mLJ [16]. In the LNS case, observe that the inference rules of $\mathsf{LNS_I}$ are the same as those of $\mathsf{LNS_G}$ with the exception of the rules for implication, which are depicted in Fig. 8.

Theorem 4.2 *Let $\mathcal{T}_{\mathsf{I}}$ contain* pos_{t4}, neg_l *plus the introduction clauses of $\mathcal{T}_{\mathsf{G}}$ with the clauses for implication substituted by the clauses in Fig. 8. Then $\mathcal{T}_{\mathsf{I}}$ is adequate w.r.t.* $\mathsf{LNS_I}$.

Proof. Observe that pos_{t4} moves left formulas from the linear context to the context $t4$

$$\dfrac{\dfrac{\dfrac{}{\vdash \Theta, c : \mathcal{T}_{\mathsf{I}}; \lfloor G\rfloor \Downarrow \lfloor G\rfloor^{\perp}}\;\mathsf{I_l} \qquad \dfrac{\vdash \Theta, c : \mathcal{T}_{\mathsf{I}}, t4 : \lfloor G\rfloor; \Gamma \Uparrow \cdot}{\vdash \Theta, c : \mathcal{T}_{\mathsf{I}}; \Gamma \Downarrow ?^{t4}\lfloor G\rfloor}\;\mathsf{R_n}, \mathsf{store}_c}{\vdash \Theta, c : \mathcal{T}_{\mathsf{I}}; \lfloor G\rfloor, \Gamma \Downarrow \exists F.(\lfloor F\rfloor^{\perp} \otimes ?^{t4}\lfloor F\rfloor)}\;\exists, \otimes}{\vdash \Theta, c : \mathcal{T}_{\mathsf{I}}; \lfloor G\rfloor, \Gamma \Uparrow \cdot}\;\mathsf{D_c}$$

Similarly, neg_l moves right formulas from the linear context to the (unbounded, local) context l. Hence we may always assume that the linear context is empty when applying a decide rule. This fact is actually not needed, but it simplifies the present proof.

Now, regarding implication, consider the following derivation

$$\dfrac{\dfrac{\dfrac{}{\vdash \Theta, c : \mathcal{T}_{\mathsf{I}}; \cdot \Downarrow \lceil A \supset B\rceil^{\perp}}\;\mathsf{I_c} \qquad \dfrac{\dfrac{\dfrac{\vdash \Theta(t4), c : \mathcal{T}_{\mathsf{I}}; \lfloor A\rfloor, \lceil B\rceil \Uparrow \cdot}{\vdash \Theta, c : \mathcal{T}_{\mathsf{I}}; \cdot \Uparrow \cdot /\!\!/^{t4} \vdash \Theta(t4), c : \mathcal{T}_{\mathsf{I}}; \cdot \Uparrow \lfloor A\rfloor \parr \lceil B\rceil}\;R_{\mathsf{r}}, \parr, 2 \times \mathsf{store}}{\vdash \Theta, c : \mathcal{T}_{\mathsf{I}}; \cdot \Uparrow \cdot /\!\!/^{t4} \vdash \cdot; \cdot \Uparrow \lfloor A\rfloor \parr \lceil B\rceil}\;{?^{t4}}_{\mathsf{4}}}{\vdash \Theta, c : \mathcal{T}_{\mathsf{I}}; \cdot \Downarrow !^{t4}(\lfloor A\rfloor \parr \lceil B\rceil)}\;!^{t4}}{\vdash \Theta, c : \mathcal{T}_{\mathsf{I}}; \cdot \Downarrow \exists F, G.(\lceil F \supset G\rceil^{\perp} \otimes !^{t4}(\lfloor F\rfloor \parr \lceil G\rceil))}\;\exists, \exists, \otimes}{\vdash \Theta, c : \mathcal{T}_{\mathsf{I}}; \cdot \Uparrow \cdot}\;\mathsf{D_c}$$

Observe that $\Theta(l)$ contains all the right formulas (that will be "forgotten") while $\Theta(t4)$ contains all the left formulas (that will be carried over the components). Hence this derivation is adequate w.r.t. the implication in $\mathsf{LNS_I}$. The other cases are similar to the case for $\mathsf{LNS_G}$. As a final remark, note that, since $t4 \preceq c$, the theory $\mathcal{T}_{\mathsf{I}}$ always move between components. $\Box$

Let us now move to the modal case. The (parameterized) clauses specifying the rules for box are given in Fig. 8. The theory $\mathcal{T}_{\Box i}$ for the modal logic resulting from extending K with the axioms in the list i is given by the clauses of $\mathcal{T}_{\mathsf{G}}$ (Fig. 6) plus the clauses neg_l and pos_l (Fig. 7) and the clauses $\Box_{Li}$ and $\Box_{Ri}$ (Fig. 8). For example, $\mathcal{T}_{\Box t4} = \mathcal{T}_{\mathsf{G}} \cup \{\mathsf{neg}_l, \mathsf{pos}_l\} \cup \{\Box_{Lt4}, \Box_{Rt4}\}$.

Theorem 4.3 *$\mathcal{T}_{\Box i}$ is adequate w.r.t.* $\mathsf{LNS}_{\mathsf{K}i}$.

Proof. The proof follows more or less the script from the intuitionistic case. Consider the following derivation.

$$\dfrac{\dfrac{\dfrac{}{\vdash \Theta, c : \mathcal{T}_{\Box i}; \cdot \Downarrow \lceil \Box A \rceil^{\perp}}\ \mathsf{I_c} \qquad \dfrac{\dfrac{\dfrac{\vdash \Theta(4), c : \mathcal{T}_{\Box i}; \lceil A \rceil \Uparrow \Theta(\mathsf{k})}{\vdash \Theta, c : \mathcal{T}_{\Box i}; \cdot \Uparrow \cdot /\!/^{i} \vdash \Theta(4), c : \mathcal{T}_{\Box i}; \cdot \Uparrow \lceil A \rceil, \Theta(\mathsf{k})}\ \mathsf{R_r}, \mathsf{store}}{\vdash \Theta, c : \mathcal{T}_{\Box i}; \cdot \Uparrow \cdot /\!/^{i} \vdash \cdot ; \cdot \Uparrow \lceil A \rceil}\ ?^{i}{}_{\mathsf{4}}, ?^{i}{}_{\mathsf{k}}}{\vdash \Theta, c : \mathcal{T}_{\Box i}; \cdot \Downarrow\ !^{i} \lceil A \rceil}\ !^{i}}{\vdash \Theta, c : \mathcal{T}_{\Box i}; \cdot \Downarrow \exists F. (\lceil \Box F \rceil^{\perp} \otimes !^{i} \lceil F \rceil)}\ \exists, \otimes}{\vdash \Theta, c : \mathcal{T}_{\Box i}; \cdot \Uparrow \cdot}\ \mathsf{D_c}$$

Observe that $\Theta(j \mid i \not\preceq j)$ contains all the right formulas, together with all the left formulas in contexts not related to i (that will be "forgotten"); $\Theta(\mathsf{4})$ contains all the formulas in $\Theta(j)$ s.t. $\mathsf{4} \in \mathcal{U}(j)$ and $\Theta(\mathsf{k})$ contains the other formulas. Hence this derivation is adequate w.r.t. the $\Box$ in $\mathsf{LNS}_{\mathsf{K}i}$. $\Box$

It is worth noticing the *modularity* of the encodings: all the modal systems have *exactly* the same encoding, only differing on the meta-level modality. This is a direct consequence of locality, granted by LNS. Therefore, we are able to spot the core characteristics of the logical systems, allowing punctual actions to be taken at the meta-level. We will profit from this widely in the next section.

We finish this section by observing that the adequacy does not reach its strongest possible level, as it is the case for the rules in $\mathcal{T}_{\mathsf{G}}$, where one focused step mimics exactly a rule application. The reason is that the search space in LNS systems is often greater than in sequent systems [14]. Hence, for example, there are no focused meta-level steps that correspond to the following valid derivation in $\mathsf{LNS_I}$

$$\dfrac{\dfrac{\dfrac{\cdot \vdash G /\!/ F, C \vdash A}{F \vdash G /\!/ C \vdash A}\ \texttt{lift}}{F \vdash G /\!/ C \vdash (A \vee B)}\ \vee_{R1}}{F \vdash G, C \supset (A \vee B)}\ \supset_{R}$$

However, the sequentialization result in [25] implies that, restricted to LNS *normal* proofs, the adequacy is achieved on its highest level. Since the results of the present paper do not depend on that, we will avoid such a discussion.

5 Cut-elimination for object logics

In this section we give a sufficient condition, *cut-coherence*, for an LNS logical system to have cut-elimination. For that, we rely on the meta-theory of SLL. Since testing cut-coherency is straightforward (see e.g., the proof of Theorem 5.5), $\mathsf{LNS}_{\mathsf{FSLL}}$ becomes a suitable logical framework for proving analyticity for a large class of systems, including several well known modal logical systems. We start by setting some requirements that OL systems should comply in order to be amenable for the SLL specification.

Requirement 5.1 (OL Syntax) *We assume that object logic's formulas are built inductively from a set of atomic propositions $\mathcal{A}$ and a set of connectives $\mathcal{C}$. We shall use $|F|$ to denote the (size) number of connectives and atomic propositions in the formula F.*

For instance, in the modal logic K, $\mathcal{C} = \{\mathtt{t}, \mathtt{f}, \wedge, \vee, \rightarrow, \Box\}$ and $|\Box A| = 1 + |A|$.

Definition 5.1 (Canonical-bipoles) *A* SLL *formula is a* bipole *[1] if no positive connective is in the scope of a negative one, bangs have negative scope while question marks have atomic scope. A* SLL *formula F is a* canonical-bipole *if F is a bipole built from* SLL *connectives and atomic formulas of the shape $\lceil A \rceil, \lfloor A \rfloor$ where A is an OL formula.*

Observe that all the clauses introducing connectives in Figs. 6, 7, 8 have the shape $\exists \overline{F}.(H^{\perp} \otimes B)$, where H is atomic and B a canonical-bipole, *e.g.* $\exists A, B.(\lceil A \supset B \rceil^{\perp} \otimes !^{t4}(\lfloor A \rfloor \parr \lceil B \rceil))$ and $\exists A.(\lfloor \Box A \rfloor^{\perp} \otimes ?^{i} \lfloor A \rfloor)$. As seen in Section 4, focusing on this kind of formulas produces specific and controlled shapes of derivations in $\mathsf{LNS}_{\mathsf{FSLL}}$.

In the following, we require that clauses encoding OL introduction rules have exactly the shape mentioned above.

Requirement 5.2 (Canonical theories and encodings) *Let $\mathcal{C}$ be the set of connectives of the object logic $\mathcal{L}$. The* encoding *of $\mathcal{L}$ as an* SLL *theory is a pair of functions $\mathcal{B}\lfloor \cdot \rfloor$ and $\mathcal{B}\lceil \cdot \rceil$ from $\mathcal{C}$ to* SLL *canonical-bipoles.*

The encoding of left and right introduction rules *for a given n-ary connective $\star \in \mathcal{C}$ is defined as, respectively*

$$\mathtt{E}\lfloor \star \rfloor = \exists F_1, ..., F_n.(\lfloor \star(F_1, ..., F_n)\rfloor^{\perp} \otimes \mathtt{B}\lfloor \star \rfloor) \qquad \mathtt{E}\lceil \star \rceil = \exists F_1, ..., F_n.(\lceil \star(F_1, ..., F_n)\rceil^{\perp} \otimes \mathtt{B}\lceil \star \rceil)$$

The canonical theory *for $\mathcal{L}$ is the least set $\mathcal{T}_{\mathcal{L}}$ s.t. (1) for each $\star \in \mathcal{C}$, $\mathtt{E}\lfloor \star \rfloor, \mathtt{E}\lceil \star \rceil \in \mathcal{T}_{\mathcal{L}}$; (2) $\mathsf{pos}_i, \mathsf{neg}_j$ may belong to $\mathcal{T}_{\mathcal{L}}$ for some subexponentials i, j; and (3) $\mathsf{init} \in \mathcal{T}_{\mathcal{L}}$ (see Fig 7).*

In words, $\mathcal{T}_{\mathcal{L}}$ includes the encoding of left and right introduction rules as well as the initial rule, and it may include the encoding of the structural rules of weakening and contraction.

Encoded inference rules determine completely the shape of meta-level derivations. In fact, focusing on $R = \exists \overline{F}.H^{\perp} \otimes B$ necessarily produces an open derivation of the form

$$\dfrac{\dfrac{\Pi}{\vdash \Theta; \Gamma' \Downarrow B}}{\vdash \Theta; \Gamma \Downarrow R}\ \exists, \otimes, \mathsf{I}$$

where $H \in \Theta$ and $\Gamma = \Gamma'$ or $\Gamma = H, \Gamma'$. Regarding Π:

(B1) it finishes with one of the rules $\top$ or 1 (with no additional premises); or

(B2) the positive phase ends with negative/exponential phases with leaves of the shape

(B2-A) $$\dfrac{\vdash \Theta, \Upsilon_1; \Gamma'_1, \Psi_1 \Uparrow \cdot \quad \cdots \quad \vdash \Theta, \Upsilon_n; \Gamma'_n, \Psi_n \Uparrow \cdot \quad \cdots}{\vdash \Theta; \Gamma' \Downarrow B}$$

(B2-B) $$\dfrac{\cdots \quad \dfrac{\dfrac{\cdots \quad \vdash \Theta(4), \Upsilon(4); \Gamma_F, \Theta(\mathsf{k}), \Upsilon(\mathsf{k}) \Uparrow \cdot}{\vdash \Theta(4), \Upsilon(4); \cdot \Uparrow F, \Theta(\mathsf{k}), \Upsilon(\mathsf{k})}}{\vdash \Theta, \Upsilon; \cdot \Uparrow \cdot /\!\!/^{i} \vdash \cdot; \cdot \Downarrow !F}}{\vdash \Theta; \Gamma' \Downarrow B}$$

Fact 5.2 *Contexts in the leaves of Π can only: shorten; and/or expand with atomic subformulas of B.*

It is well known that bipoles are totally decomposed into its atomic components during a focused phase (please refer to, *e.g.*, [17] or [21] for the proof). Hence, $\Psi_i, \Upsilon_i, \Upsilon$ contain only atomic subformulas of B. In (B2-A), Γ' is split (multiplicative case) or shared (additive case) on the premises (Γ'_i). In (B2-B), $\Theta(4), \Upsilon(4)$ contain all the formulas in $\Theta(j), \Upsilon(j)$ s.t. $4 \in \mathcal{U}(j)$, $\Theta(\mathsf{k}), \Upsilon(\mathsf{k})$ contain (atomic) formulas in $\Theta(j)$ s.t. $i \preceq j$ and Γ_F contains only atomic subformulas of F.

The case (B1) embodies, e.g., the encoding of falsity ($\mathtt{f}$), while the case (B2-A) reflects the encoding of the introduction rules for the connectives $\wedge, \vee$ and $\rightarrow$ in Fig. 6. The case (B2-B) typifies rules like $\Box_R$ and $\supset_R$ in Fig. 8. The reader may compare the derivations in the previous section with the above cases. It is easy to see that the resulting bipoles in each of the presented encodings falls unequivocally on one of these cases.

Finally, in order to guarantee that the encoding actually reflects the specified OL, we need the following requirement. Such adequacy results were already proved in Section 4 for the logics studied here.

Requirement 5.3 (Adequacy) *Let $\mathcal{T}_{\mathcal{L}}$ be the canonical theory for the OL $\mathcal{L}$. We assume that the OL sequent $\Gamma \vdash \Delta$ is provable in $\mathcal{L}$ iff the sequent $\vdash c : \mathcal{T}_{\mathcal{L}}; \lfloor \Gamma \rfloor, \lceil \Delta \rceil \Uparrow \cdot$ is provable in $\mathsf{LNS}_{\mathsf{FSLL}}$.*

5.1 Cut-coherence and cut-elimination

The OL cut-rule can be specified as the bipole $\mathsf{cut} = \exists F.(\lfloor F \rfloor \otimes \lceil F \rceil)$. In fact, focusing on that formula mimics exactly the behavior of the cut-rule at the object level:

$$\dfrac{\Gamma_1 \vdash \Delta_1, F \quad \Gamma_2, F \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathsf{cut} \quad \Longleftrightarrow \quad \dfrac{\dfrac{\vdash c : \mathsf{cut}; \lfloor \Gamma_1 \rfloor, \lceil \Delta_1 \rceil, \lceil F \rceil \Uparrow \cdot \quad \vdash c : \mathsf{cut}; \lfloor \Gamma_2 \rfloor, \lfloor F \rfloor, \lceil \Delta_2 \rceil \Uparrow \cdot}{\vdash c : \mathsf{cut}; \lfloor \Gamma_1 \rfloor, \lfloor \Gamma_2 \rfloor, \lceil \Delta_1 \rceil, \lceil \Delta_2 \rceil \Downarrow \mathsf{cut}}\ \exists, \otimes, \mathsf{R_n}, \mathsf{store}}{\vdash c : \mathsf{cut}; \lfloor \Gamma_1 \rfloor, \lfloor \Gamma_2 \rfloor, \lceil \Delta_1 \rceil, \lceil \Delta_2 \rceil \Uparrow \cdot}\ \mathsf{D_c}$$

We shall use cut_n to denote the rule cut applied to (OL) formulas of size strictly smaller than n. For instance, if $G = G_1 \star G_2$, a valid application of $\mathsf{cut}_{|G|}$ can instantiate the existentially quantifier variable F in cut with either G_1 or G_2 (but not with G).

Using the formulas cut and init, we can prove that $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ are *duals* .

Fact 5.3 ([17]) *The following sequent is provable in $\mathsf{LNS}_{\mathsf{FSLL}}$: $\vdash c : \mathsf{cut}, c : \mathsf{init}; \Uparrow \lfloor F \rfloor \equiv \lceil F \rceil^{\perp}$.*

This duality can be tailored to the right and left bodies (see Requirement 5.2) of OL's rules as well.

Definition 5.4 (Cut-coherence) *Let $\mathcal{T_L}$ be the canonical theory of the OL $\mathcal{L}$. We say that $\mathcal{T_L}$ is* cut-coherent *if, for each connective $\star \in \mathcal{C}$, the sequent below is provable*

$$\vdash c : \mathsf{cut}_{|F|}; \Uparrow \forall F_1, ..., F_n.((\mathcal{B}\lfloor\star\rfloor)^\perp \parr (\mathcal{B}\lceil\star\rceil)^\perp)$$

Theorem 5.5 *All the encodings in Section 4 are cut-coherent.*

Proof. Let us show some cases. The following two derivations show the cut-coherence for $\supset$ in the system $\mathsf{LNS_I}$ and also for $\Box$ in the logic K (see Fig. 8). In order to simplify the notation, we use cut_A to denote the derivation resulting after focusing on the instance of $\mathsf{cut} = \exists F.(\lfloor F\rfloor \otimes \lceil F\rceil)$ with the subformula A (thus using the rules for $\exists$, $\otimes$ and I).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash c : \mathsf{cut}, t4 : \lfloor A\rfloor^\perp \otimes \lceil B\rceil^\perp; \lfloor A\rfloor, \lceil B\rceil \Uparrow \cdot}\ \mathsf{D_c}, \otimes, \mathsf{I_l}}{\vdash c : \mathsf{cut}, t4 : \lfloor A\rfloor^\perp \otimes \lceil B\rceil^\perp; \lceil A\rceil^\perp, \lfloor B\rfloor^\perp \Uparrow \cdot}\ \mathsf{cut}_A, \mathsf{cut}_B}{\vdash c : \mathsf{cut}; \cdot \Uparrow \lceil A\rceil^\perp \parr \lfloor B\rfloor^\perp, ?^{t4}(\lfloor A\rfloor^\perp \otimes \lceil B\rceil^\perp)}\ \parr, ?}{\vdash c : \mathsf{cut}; \cdot \Uparrow \forall A, B.(\lceil A\rceil \otimes \lfloor B\rfloor)^\perp \parr ((!^{t4}(\lfloor A\rfloor \parr \lceil B\rceil))^\perp)}\ \forall, \parr$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash c : \mathsf{cut}; \lceil A\rceil^\perp, \lceil A\rceil \Uparrow \cdot}\ \mathsf{D_l}, \mathsf{I_l}}{\vdash c : \mathsf{cut}; \lfloor A\rfloor^\perp, \lceil A\rceil^\perp \Uparrow \cdot}\ \mathsf{cut}_A}{\vdash c : \mathsf{cut}; \cdot \Uparrow \lfloor A\rfloor^\perp, \lceil A\rceil^\perp}\ \mathsf{store}}{\vdash c : \mathsf{cut}, k : \lceil A\rceil^\perp; \cdot \Uparrow \cdot /\!/^k \vdash c : \mathsf{cut}; \cdot \Uparrow \lfloor A\rfloor^\perp, \lceil A\rceil^\perp}\ \mathsf{R_r}}{\vdash c : \mathsf{cut}, k : \lceil A\rceil^\perp; \cdot \Uparrow \cdot /\!/^k \vdash \cdot ; \cdot \Uparrow \lfloor A\rfloor^\perp}\ ?^k, ?^c}{\vdash c : \mathsf{cut}, k : \lceil A\rceil^\perp; \cdot \Downarrow\, !^k \lfloor A\rfloor^\perp}\ !^k}{\vdash c : \mathsf{cut}, k : \lceil A\rceil^\perp; !^k \lfloor A\rfloor^\perp \Uparrow \cdot}\ \mathsf{D_l}}{\vdash c : \mathsf{cut}; \cdot \Uparrow \forall A.((\lfloor \Box A\rfloor^\perp \otimes ?^k \lfloor A\rfloor)^\perp \parr (\lceil \Box A\rceil^\perp \otimes !^k \lceil A\rceil)^\perp)}$$

The cases for the other modal rules are similar (using the appropriate subexponential). Note that in the case of constants/units, the rule cut cannot be used (since for a constant a, $|a| = 1$ and there are no OL formulas of size 0). This reflects the intuition that the cut-elimination procedure for constants cannot rely on induction on subformulas of that connective. Consider the unit $\mathtt{f}$ that only has a left rule (Fig. 6). Hence, the right rule is specified as $\lfloor \mathtt{f}\rfloor^\perp \otimes 0$ (there is no introduction rule for 0 in LL). Note that $\top^\perp = 0$ and then, the bodies of those rules are indeed cut-coherent (but cut is not needed in that proof). $\square$

Now we are ready to state the main result: given two cut-free proofs (from the object-level point of view, using only the theory $\mathcal{T_L}$) introducing the cut formula F, it is possible to prove the same sequent using the rule cut (at the object-level) with strict subformulas of F ($\mathsf{cut}_{|F|}$).

Theorem 5.6 *Let $\mathcal{T_L}$ be the theory of a given OL $\mathcal{L}$ and Γ, Δ, Ψ be multisets of atoms of the form $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$. If the sequents $\vdash c : \mathcal{T_L}, \Gamma; \Delta \Downarrow \lfloor F\rfloor$ and $\vdash c : \mathcal{T_L}, \Gamma; \Psi \Downarrow \lceil F\rceil$ are both provable then the sequent $\vdash c : \mathcal{T_L}, c : \mathsf{cut}_{|F|}, \Gamma; \Delta, \Psi \Uparrow \cdot$ is also provable.*

Proof. We know that both $\vdash c : \mathcal{T_L}, \Gamma; \Delta, \lfloor F\rfloor \Uparrow$ and $\vdash c : \mathcal{T_L}, \Gamma; \Psi, \lceil F\rceil \Uparrow$ are provable (since focusing is lost in both $\lfloor F\rfloor$ and $\lceil F\rceil$). Call these proofs $[\Sigma]$ and $[\Pi]$. Since Γ, Δ, Ψ only contain atoms that cannot get focus, the proof of such sequents must start with an application of the decision rule on one of the formulas in $\mathcal{T_L}$. We proceed by induction on the height of the derivations $[\Sigma]$ and $[\Pi]$. We have several cases.

Non-principal cases. If $[\Sigma]$ starts with a right rule we have 3 cases. Note that $\lfloor F\rfloor$ (a left atom) cannot be the head of that rule. The case $(B1)$ is trivial. The case (B2-A) follows by induction. Consider for instance a derivation with only two premises. Note that $\lfloor F\rfloor$ can go to one or both premises depending whether a multiplicative or an additive connective is used. Here we consider the multiplicative case. The reduction is:

$$\dfrac{\overset{[\Xi]}{\vdash c : \mathcal{T_L}, \Gamma; \Delta_1, \lfloor F\rfloor \Uparrow \cdot} \quad \vdash c : \mathcal{T_L}, \Gamma; \Delta_2 \Uparrow \cdot}{\vdash \mathcal{T_L}, \Gamma; \Delta, \lfloor F\rfloor \Downarrow \mathtt{E}[\![\star]\!]} \rightsquigarrow \dfrac{\dfrac{\overset{[\Xi']}{\vdash c : \mathcal{T_L}, \Gamma; \Psi, \Delta_1 \Uparrow \cdot} \quad \vdash c : \mathcal{T_L}, \Gamma; \Delta_2 \Uparrow \cdot}{\vdash c : \mathcal{T_L}, \Gamma, \mathsf{cut}_{|F|}; \Delta, \Psi \Downarrow \mathtt{E}[\![\star]\!]}}{\vdash c : \mathcal{T_L}, \Gamma, \mathsf{cut}_{|F|}; \Delta, \Psi \Uparrow \cdot}$$

Derivation $[\Xi']$ results from induction on $[\Xi]$ and $[\Pi]$. The case of a left premise on the presence of $\lceil F\rceil$ is similar. Now consider the case (B2-B) and the following derivation

$$\dfrac{\overset{[?]}{\vdash c:\mathcal{T}_{\mathcal{L}},\Gamma;\lfloor F\rfloor \Uparrow \cdot /\!/ \vdash c:\mathcal{T}_{\mathcal{L}},\Gamma';\Gamma'' \Uparrow \cdot}}{\vdash c:\mathcal{T}_{\mathcal{L}},\Gamma;\lfloor F\rfloor \Downarrow \mathtt{B}\lceil\star\rceil}$$

Since rule $\mathsf{R_r}$ requires the linear context to be empty, this derivation cannot actually happen. Hence, $\lfloor F\rfloor$ cannot be principal if a creation rule is applied: it must be moved before the application of the rule to the classical context (using $\mathtt{pos}$) to be later "erased" in the penultimate component.

There are also non-principal cases where a left rule is applied but $\lfloor F\rfloor$ is not principal (similar for right rules and $\lceil F\rceil$). The procedure is similar to the one described above.

Principal cases. Now consider the case where $\lfloor F\rfloor$ and $\lceil F\rceil$ are principal in both premises thus using, respectively, the left and right introduction rules for the same connective. This case is solved by using weakening, cut-coherence and the cut-rule of linear logic:

$$\text{if}\quad \dfrac{\dfrac{\vdash c:\mathcal{T}_{\mathcal{L}},\Gamma;\Delta \Downarrow \mathtt{B}\lfloor\star\rfloor}{\vdash c:\mathcal{T}_{\mathcal{L}},\Gamma;\Delta,\lfloor F\rfloor \Downarrow \mathtt{E}\lfloor\star\rfloor}}{\vdash c:\mathcal{T}_{\mathcal{L}},\Gamma;\Delta,\lfloor F\rfloor \Uparrow \cdot}\quad \text{and}\quad \dfrac{\dfrac{\vdash c:\mathcal{T}_{\mathcal{L}},\Gamma;\Psi \Downarrow \mathtt{B}\lceil\star\rceil}{\vdash c:\mathcal{T}_{\mathcal{L}},\Gamma;\Psi,\lceil F\rceil \Downarrow \mathtt{E}\lceil\star\rceil}}{\vdash c:\mathcal{T}_{\mathcal{L}},\Gamma;\Psi,\lceil F\rceil \Uparrow \cdot}\quad \text{then}$$

$$\dfrac{\vdash c:\mathcal{T}_{\mathcal{L}},\Gamma;\Psi \Downarrow \mathtt{B}\lceil\star\rceil \qquad \dfrac{\vdash c:\mathcal{T}_{\mathcal{L}},\Gamma;\Delta \Downarrow \mathtt{B}\lfloor\star\rfloor \qquad \dfrac{}{\vdash c:\mathcal{T}_{\mathcal{L}},\Gamma,c:\mathsf{cut}_{|F|};\cdot \Uparrow (\mathtt{B}\lceil\star\rceil)^{\perp},(\mathtt{B}\lfloor\star\rfloor)^{\perp}}\ \text{cut-coherence}}{\vdash c:\mathcal{T}_{\mathcal{L}},\Gamma,c:\mathsf{cut}_{|F|};\Delta \Uparrow (\mathtt{B}\lceil\star\rceil)^{\perp}}\ \mathsf{FSLL-cut}}{\vdash c:\mathcal{T}_{\mathcal{L}},\Gamma,c:\mathsf{cut}_{|F|};\Delta,\Psi \Uparrow \cdot}\ \mathsf{FSLL-cut}$$

□

Since for every OL formula F, $|F| > 0$, by induction we conclude the following.

Corollary 5.7 (OL cut-elimination) *Let $\mathcal{T}_{\mathcal{L}}$ be the theory of a given OL $\mathcal{L}$ and Γ,Δ,Ψ be multisets of atoms of the form $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$. The sequent $\vdash c:\mathcal{T}_{\mathcal{L}},c:\mathsf{cut},\Gamma;\Delta \Uparrow \cdot$ is provable iff $\vdash c:\mathcal{T}_{\mathcal{L}},\Gamma;\Delta \Uparrow \cdot$ is provable.*

Finally, we observe that, as in [17], determining whether or not an OL encoding is cut-coherent is decidable, with the proof search in SLL bounded by the depth $v+3$ where v is the maximum number of premise atoms in the bodies of the introduction clauses.

6 Discussion and conclusion

In this paper, we have extended the sufficient criterion for cut-elimination of object logics given in [17]. For that, we moved from LL to a variant of SELL (linear logic with subexponentials), where different modal behaviors were embodied into the subexponential connectives (substructural simply dependent multimodal system SLL). This allowed to establish a *simple* yet powerful criterion –*cut-coherence* – for proving analyticity for a large class of sequent-based systems. What this criterion reflects is the *duality* of rules. In fact, checking cut-coherence is equivalent to checking $\mathtt{B}\lfloor\star\rfloor^{\perp} \equiv \mathtt{B}\lceil\star\rceil$, and vice versa. And *this* is the spirit of cut-elimination.

We start the discussion by exploring the differences between this work and the one in [21]. First of all, encoding modalities different from those in LL is really tricky (or even impossible) using SELL. For instance, the modal behavior for K itself cannot be captured in SELL, while the one for S4 can, with a very clever subexponential signature: $\langle\{l,r,\Box_L,\diamond_R,e_l,e_r,\infty\},\{r\prec\diamond_R\prec\infty, l\prec\Box_L\prec\infty, e_l\prec\diamond_R, e_l\prec\Box_L, e_r\prec\Box_L\}\rangle$, where e_l, e_r are *dummy* subexponentials. The complexity of these encodings is transferred to the half-page cut-coherence criterion presented in [21]. If we classify the results according to: (1) meta-level expressivity; and (2) cut-elimination criteria then

(1) SLL is strictly greater than SELL. In fact, although in this work we consider only unbounded subexponentials, the exact same reasoning can be done for the bounded case by simply adjusting some of the inference rules in SLL. Hence all SELL encodings shown in [21] can be transported to SLL.

(2) The cut-elimination results in this work do not extend the ones in [21] (as we are focusing on a particular class of subexponentials), neither the opposite (due to (1) reflected, *e.g.*, by the fact that K cannot be handled in [21]). What we have shown is that, for such a class, the simplicity of [17] is recovered while, at the same time, subexponentials are handled in a very natural way. This brought back to the spot the core of cut-elimination: the *duality* of inference rules.

The key feature for achieving all this is *modularity*. Everywhere. Starting from the choice of LNS, a generalization of sequent systems, as the base framework. This allows for the *locality* of rules, enabling the central behavior of connectives to be shared among several different logics and leaving to the subexponentials the work of separating modal behaviors. That is, modalities reflect modalities, while (vanilla) LL captures rules as rewriting clauses (as it should be). Second, structural rules are parametric w.r.t. subexponentials, allowing for a clear separation between modals and local structural behaviors. And last, but not least, since subexponentials in SLL also reflect Kripke models, logics having the same semantic behavior share the same modal characteristics. This is the case, *e.g.*, for intuitionistic logic and S4.

Analyticity is attached to the *logical system*, not to the logic itself. So one could argue that we were, in fact, just changing the initial problem. This would be the case if we would have adopted, *e.g.*, nested systems instead of LNS. Indeed, there are cut-free nested systems for modal logics B and S5 [3], for example, while there is no known simple cut-free sequent systems for such logics. However, in [25] a class of nested systems that can be sequentializable into sequent systems was determined, and LNS systems, being a special case of nested systems with trees replaced by lines, fall into this class. Hence the cut-elimination criterion for LNS presented here is transferred to the respective sequent framework.

Logical frameworks, based on type systems, have also been used for characterizing and proving cut-elimination theorems of object-logics (see e.g., [15, 23]). Usually, the embedding of the OL into the logical framework is not simple/direct. The approach followed here is rather different: we provide easy-to-check conditions that guarantee that the property holds. In the near future, we plan to formalize our results in Coq, as done in [6]. Also, it would be interesting to analyze the case of non-normal modal logics [14]. Finally, it would be interesting to explore the failure cases: is it possible, at the meta-level, to identify the reasons for the lack of analyticity? This would push the line of investigation towards finding *necessary* conditions for cut-elimination.

References

[1] Andreoli, J.-M., *Logic programming with focusing proofs in linear logic*, Journal of Logic and Computation **2** (1992), pp. 297–347.

[2] Avron, A., *The method of hypersequents in the proof theory of propositional non-classical logics*, in: *Logic: from foundations to applications: European logic colloquium*, Clarendon Press, 1996 pp. 1–32.

[3] Brünnler, K., *Deep sequent systems for modal logic*, Archive for Mathematical Logic **48** (2009), pp. 551–577.
URL http://link.springer.com/article/10.1007/s00153-009-0137-3

[4] Danos, V., J.-B. Joinet and H. Schellinx, *The structure of exponentials: Uncovering the dynamics of linear logic proofs*, in: G. Gottlob, A. Leitsch and D. Mundici, editors, *Kurt Gödel Colloquium*, LNCS **713** (1993), pp. 159–171.

[5] Demri, S., *Complexity of simple dependent bimodal logics*, in: R. Dyckhoff, editor, *TABLEAUX 2000*, LNCS **1847**, Springer, 2000 pp. 190–204.

[6] Felty, A., C. Olarte and B. Xavier, *A focused linear logical framework and its application to meta theory of object logics* (2020), submitted to MSCS. https://github.com/meta-logic/coq-fll.

[7] Gentzen, G., *Investigations into logical deduction*, in: M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, North-Holland, 1969 pp. 68–131.

[8] Girard, J.-Y., *Linear logic*, Theoretical Computer Science **50** (1987), pp. 1–102.

[9] Guerrini, S., S. Martini and A. Masini, *An analysis of (linear) exponentials based on extended sequents*, Logic Journal of the IGPL **6** (1998), pp. 735–753.
URL https://doi.org/10.1093/jigpal/6.5.735

[10] Kanovich, M. I., S. Kuznetsov, V. Nigam and A. Scedrov, *Subexponentials in non-commutative linear logic*, Math. Struct. Comput. Sci. **29** (2019), pp. 1217–1249.
URL https://doi.org/10.1017/S0960129518000117

[11] Lellmann, B., *Linear nested sequents, 2-sequents and hypersequents*, in: *24th TABLEAUX*, 2015, pp. 135–150.

[12] Lellmann, B., C. Olarte and E. Pimentel, *A uniform framework for substructural logics with modalities*, in: *LPAR-21*, 2017, pp. 435–455.
URL http://www.easychair.org/publications/paper/340350

[13] Lellmann, B. and E. Pimentel, *Proof search in nested sequent calculi*, in: *LPAR-20*, 2015, pp. 558–574.
URL http://dx.doi.org/10.1007/978-3-662-48899-7_39

[14] Lellmann, B. and E. Pimentel, *Modularisation of sequent calculi for normal and non-normal modalities*, ACM Transactions on Computational Logic (TOCL) **20** (2019), p. 7.

[15] Licata, D. R., M. Shulman and M. Riley, *A fibrational framework for substructural and modal logics*, in: D. Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, LIPIcs **84** (2017), pp. 25:1–25:22.
URL https://doi.org/10.4230/LIPIcs.FSCD.2017.25

[16] Maehara, S., *Eine darstellung der intuitionistischen logik in der klassischen*, Nagoya Mathematical Journal (1954), pp. 45–64.

[17] Miller, D. and E. Pimentel, *A formal framework for specifying sequent calculus proof systems*, Theoretical Computer Science **474** (2013), pp. 98–116.
URL http://dx.doi.org/10.1016/j.tcs.2012.12.008

[18] Miller, D. and A. Saurin, *From proofs to focused proofs: a modular proof of focalization in linear logic*, in: *CSL*, LNCS **4646**, 2007, pp. 405–419.

[19] Nigam, V. and D. Miller, *A framework for proof systems*, J. of Automated Reasoning **45** (2010), pp. 157–188.
URL http://springerlink.com/content/m12014474287n423/

[20] Nigam, V., C. Olarte and E. Pimentel, *On subexponentials, focusing and modalities in concurrent systems*, Theor. Comput. Sci. **693** (2017), pp. 35–58.
URL https://doi.org/10.1016/j.tcs.2017.06.009

[21] Nigam, V., E. Pimentel and G. Reis, *An extended framework for specifying and reasoning about proof systems*, Journal of Logic and Computation **26** (2016), pp. 539–576.
URL http://dx.doi.org/10.1093/logcom/exu029

[22] Olarte, C., D. Chiarugi, M. Falaschi and D. Hermith, *A proof theoretic view of spatial and temporal dependencies in biochemical systems*, Theor. Comput. Sci. **641** (2016), pp. 25–42.
URL https://doi.org/10.1016/j.tcs.2016.03.029

[23] Pfenning, F., *Structural cut elimination: I. intuitionistic and classical logic*, Inf. Comput. **157** (2000), pp. 84–141.
URL https://doi.org/10.1006/inco.1999.2832

[24] Pimentel, E., *A semantical view of proof systems*, in: *Logic, Language, Information, and Computation - 25th International Workshop, WoLLIC 2018, Bogota, Colombia, July 24-27, 2018, Proceedings*, 2018, pp. 61–76.
URL https://doi.org/10.1007/978-3-662-57669-4_3

[25] Pimentel, E., R. Ramanayake and B. Lellmann, *Sequentialising nested systems*, in: *TABLEAUX 2019*, 2019, pp. 147–165.
URL https://doi.org/10.1007/978-3-030-29026-9_9

[26] Poggiolesi, F., *The method of tree-hypersequents for modal propositional logic*, in: *Towards Mathematical Philosophy*, Trends In Logic **28**, Springer, 2009 pp. 31–51.

[27] Straßburger, L., *A local system for linear logic*, in: *Proceedings of LPAR 2002*, number 2514 in LNCS, 2002, pp. 388–402.

[28] Viganò, L., "Labelled non-classical logics," Kluwer, 2000.

[29] Wansing, H., *Sequent systems for modal logics*, in: D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Vol.8*, Springer-Verlag Berlin Heidelberg, 2002 .
URL http://link.springer.com/chapter/10.1007%2F978-94-010-0387-2_2

[30] Xavier, B., C. Olarte, G. Reis and V. Nigam, *Mechanizing focused linear logic in coq*, in: S. Alves and R. Wasserman, editors, *LSFA 2017*, ENTCS **338** (2017), pp. 219–236.
URL https://doi.org/10.1016/j.entcs.2018.10.014

ReLo: a dynamic logic to reason about Reo circuits [1]

Erick Grilo[2]

Departamento de Ciência da Computação
Universidade Federal Fluminense
Niterói, Brazil

Bruno Lopes[3]

Departamento de Ciência da Computação
Universidade Federal Fluminense
Niterói, Brazil

Abstract

Critical systems may require high reliability and are present in many domains; in many cases, their failure resulted in financial damage or even loss of lives. Standard software engineering techniques are not designed to ensure the absence of unacceptable failures and/or that crucial requirements are fulfilled. Tools based on formal methods aim to tackle this issue. Reo is a graphical coordination model that focuses on interactions between systems. Its design is tailored to denote natural properties in distributed systems, such as remote function calls and message passing. This paper proposes *ReLo*, a dynamic logic tailored to reason about Reo models. Syntax and semantics are presented with an axiomatization followed by a usage example.

Keywords: Reo, Dynamic logic, Modal logic

1 Introduction

With the advent of the information age, as computers play a key role in most of society, new technologies and techniques have emerged in computer science's various fields. In software development, service-oriented computing [14] and model-driven development [4] are examples of these techniques, where the first advocates computing based on preexisting systems (services) as described by Service-oriented architecture (SOA), and the latter is a development technique which considers the implementation of a system based on a model. Researchers also have applied approaches such as formal methods in software development to formalize and assure that certain (critical) systems have some required properties [9,13].

A prominent modelling language is Reo [1], a language bound to externally coordinate how interconnected systems communicate between themselves. Models in Reo are compositionally built from connectors, where each connector in Reo denotes a specific communication behavior. Reo has proven to be successful in modeling the orchestration of concurrent systems' interaction, being employed in a wide range of applications, from process modeling to Web-Services integration [3,12] and even model-checking [11].

The development of systems based on SOA employing model-driven development has been proved to be a valuable approach [6], considering the advantages of reusing software can bring (such as cost reduction). By formally validating models used as the basis for the development of these systems, it is possible to detect and avoid errors which could be detected only in posterior phases of software development, or even in a productive environment, avoiding unwanted costs and losses.

[1] The authors would like to thank FAPERJ, CAPES and CNPq for supporting this research

[2] Email: simas_grilo@id.uff.br

[3] Email: bruno@ic.uff.br

In this paper, we propose a logical approach to reason about Reo connectors. Among the many formal semantics for Reo [8], as far as the authors are concerned there is no logic specifically designed to model and reason directly about Reo circuits. The advantages of our approach include the natural modelling of Reo models in a logical language, not requiring any translation process to any intermediate formalisms, and a Coq prototypical implementation of the logic's core aspects (currently under development at https://github.com/frame-lab/ReoLogicCoq), enabling the modelling, reasoning and the extraction of code regarding the connector, after its properties have been ensured.

This paper is structured as follows. Sec 2 briefly introduces Reo modelling language and its graphical tooling. Sec. 3 introduces our approach, discussing its core definitions, while Sec. 4 explores a usage example by reasoning on some formalized properties, and Sec. 5 closes our discussion pointing future and ongoing directions regarding the presented work.

2 Reo

Reo [1,2] is a graphical coordination model based on channels where complex coordinators are compositionally built from simpler ones. These complex coordinators are called connectors and constitute the very heart of the Reo modeling language. Reo has as its main objective to be a language that connects instances of different components that act together in a component-based system, coordinating how such communication takes place.

Channels in Reo are defined as a point-to-point link between two distinct nodes, where each channel has its behavior based on distributed systems' communication, such as buffering or (a)synchronous messaging. Such channels have exactly two ends: source end as the entry point of data into the channel, and the sink end as the exit point for data to leave the channel. Channels may be used to compose more complex connectors using a composition operation, such as the one provided by Baier et al. [5], which may consider both canonical Reo connectors and user-defined connectors. Fig. 1 shows the basic set of connectors as Kokash et al. [10] present.

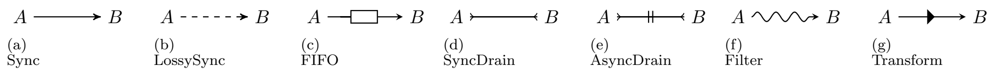


Fig. 1. Canonical Reo connectors

The intuitive interpretation of the channels of Fig. 1 is as follows. The Sync channel models the synchronous communication between two entities, while LossySync models the same behavior, also enabling the modelling of scenarios where the data is lost in the process. The FIFO connector models a buffer-like communication pattern. Channels SyncDrain and AsyncDrain respectively synchronize and desynchronize data flows in the connected entities. Lastly, the Filter channel filters data flow given a specific property P a data item may satisfy, and the Transform channel transforms the data flow from A to B with a transformation function f.

Reo plays a central role in integrating software components, especially considering domains where such components are developed focusing on their primary objectives, not expressing any concern regarding external systems, as Component-Based Software Engineering advocates. The resulting system is created through the orchestrated interaction of these software components, where Reo may be adequate in orchestrating such interaction.

We introduce Fig. 2 depicting a Reo connector named Sequencer [4] which models the data flow between three entities in a sequential manner. The black bullet denotes a data item which will be sequentially transmitted through A, B, and C. The Sequencer can be used to model scenarios where processes attached to the port names need to be sequentialized (e.g., the black bullet may denote a token which is alternatively transmitted to A,B and C, and some executing program needs this token to end its execution).

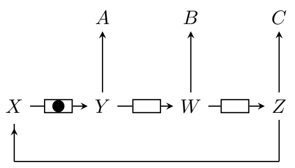


Fig. 2. Modelling of the Sequencer in Reo

[4] http://reo.project.cwi.nl/v2/#examples-of-complex-connectors

3 A logic for Reo

ReLo is a logic tailored to model and reason about Reo circuits. Although there are many semantics for Reo [8], the logic framework *ReLo* proposes benefits from a simple Kripke structure that can naturally model Reo connectors, specify properties and verify these properties. It is a formal system that subsumes Reo's semantics (its connectors' behaviour) to reason over them.

In what follows, we detail *ReLo*'s language, semantics, and present an axiomatic system, along with examples regarding the Reo model portrayed by Fig. 2. In this section, we introduce the main definitions regarding language and semantics from Sec. 3.1 to 3.5, and briefly discuss *ReLo*'s axiomatic system in Sec. 3.6.

3.1 Language and semantics

Definition 3.1 [Language] *ReLo*'s language consists of:

- An enumerable set of propositions Φ
- Canonical Reo programs (denoting connectors in the set presented in Fig. 1)
- A set of port names $\mathcal{N}$
- A program composition symbol: $\odot$
- A sequence t denoting data flow on ports p
- Program iteration operator: *

A data marking t describes how a given port name denoting an entity of a Reo circuit p interfaces with the model by depicting its data flow. The symbol ϵ denotes no data flow in the whole circuit. Therefore, t defines the actual data flow of the whole circuit, and it can be described by the following grammar. Data flow $\langle portName\rangle\langle data\rangle$ indicates that there is a data item in $\langle portName\rangle$, while $\langle portName\rangle\langle data\rangle\langle portName\rangle$ is the case where data is into a buffer between two ports. We define Reo programs as the Reo connectors written in *ReLo*'s language. By considering data flow into modalities we increase the expressibility of the language, enabling the possibility of modelling particular executions of a program.

$\langle t\rangle ::= \langle t\rangle\ \langle flow\rangle \mid \langle flow\rangle \mid \epsilon \mid \langle flow\rangle ::= \langle portName\rangle\ \langle data\rangle \mid \langle portName\rangle\ \langle data\rangle\ \langle portName\rangle \mid \langle portName\rangle ::= p \in \mathcal{N} \mid \langle data\rangle ::= 0 \mid 1$

We define formulae in *ReLo* as follows: $\phi = p \mid \top \mid \neg\phi \mid \phi \wedge \psi \mid \langle t, \pi\rangle\phi$, such that $p \in \Phi$. We use the standard abbreviations $\top \equiv \neg\bot, \phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi), \phi \to \psi \equiv \neg\phi \vee \psi$ and $[t, \pi] \equiv \neg\langle t, \pi\rangle\neg\phi$. In what follows, we define the notion of Frame and Model. We use the notation $t \prec t'$ to depict that all data flows in t are also in t'.

Definition 3.2 [Frame] A frame is formally defined as a tuple $\mathcal{F} = \langle S, \Pi, R_\Pi, \delta, \lambda\rangle$, where each element of $\mathcal{F}$ is described as follows.

- S is a non-empty enumerable set of states.
- A Reo program Π.
- $R_\Pi \subseteq S \times S$ is a relation defined as follows.
  - $R_{\pi_i} = \{uR_{\pi_i}v \mid f(t, \pi_i) \prec \delta(v),\ t \prec \delta(u)$ and π_i is any combination of any atomic program which is a subprogram of Π.
  - $R_{\pi_i^\star} = R^\star_{\pi_i}$, the reflexive transitive closure (RTC) of R_{π_i}.
- $\lambda\colon S \times \mathcal{N} \to \mathbb{R}$ is a function that returns the time instant a data item in a data markup flows through a port name.
- $\delta\colon S \to T$, is a function that returns data in ports of the circuit in a state $s \in S$.

Definition 3.3 [Model] $\mathcal{M} = \langle \mathcal{F}, \mathbf{V}\rangle$, where $V\colon S \to 2^\Phi$ is a model's valuation function

3.2 Parsing of a Reo program π: parse

The function $parse(\pi, s)$ is defined as an auxiliary definition which interprets a Reo program π and dismembers it in a sequence of flows s, which will later be used to process an input t for π. We consider that each non-canonical Reo program π is the composition of programs $\pi_i \odot \pi_j, \pi_i, \pi_j$ are Reo programs. The operator $\circ$ adds a data flow to a sequence of data flows s. Due to lack of space, we omit part of $parse$'s definition. Its definition for the remainder of the connectors follows a similar idea, considering their respective particularities and are available in the project's repository.

$$parse(\pi_i \odot \pi_j, s) = \begin{cases} parse(\pi_j, s \circ A \to B), \text{ iff } \pi_i = \; A \longrightarrow B \\ \quad (s \circ A \to B), \text{ iff } \pi_i \odot \pi_j = \pi_i \\ parse(\pi_j, s) \circ fifo(A,B), \text{ iff } \pi_i = \; A \text{—}\!\boxed{}\!\!\longrightarrow B \\ \quad (s \circ fifo(A,B)), \text{ iff } \pi_i \odot \pi_j = \pi_i \\ parse(\pi_j, SBlock(A,B) \circ s), \text{ iff } \pi_i = \; A \rightarrowtail\!\!\!\!\leftarrowtail B \\ \quad (SBlock(A,B) \circ s), \text{ iff } \pi_i \odot \pi_j = \pi_i \end{cases} \tag{1}$$

We employ *parse* in order to interpret Reo programs π as a sequence of occurrences of possible data flow (where each flow corresponds to the execution of a Reo program composing π). These data flow may be direct (of the form $A \to B$), flow "blocks" induced by connectors such as SyncDrain and aSyncDrain (the first one requires that data flow synchronously through its ports, while the latter requires that data flow asynchronously through its ports), or the notion of a buffer introduced by FIFO connectors, which data flow into a buffer before flowing out of the channel. There are also special data flows which denote the transformation and the filtering of data flow from A to B, respectively $Transform(f, A, B)$ and $Filter(P, A, B)$ ($f\colon Data \to Data$ and P a proposition over A's data item) denoting Transform and Filter connectors' behaviour.

Example 3.4 shows how *parse* functions and illustrates why it is necessary. The programs that depict the FIFO connectors from Fig. 2 are the last programs to be executed, while the ones that denote "immediate" flow (the Sync channels) come first. This is done to preserve the data when these connectors fire (if eligible). Suppose that there is a data item in the buffer between X and Y and a data item in Y (i.e., $t = X1Y, Y0$). If the data item leaves the buffer first than the data item in Y, the latter will be overwritten and the information is lost.

Example 3.4 let π be the Reo program corresponding to the circuit in Fig. 2:
$\pi = \; \mathrm{X} \text{—}\!\boxed{}\!\!\longrightarrow \mathrm{Y} \odot \mathrm{Y} \longrightarrow \mathrm{A} \odot \mathrm{Y} \text{—}\!\boxed{}\!\!\longrightarrow \mathrm{W} \odot \mathrm{W} \longrightarrow \mathrm{B} \odot \mathrm{W} \text{—}\!\boxed{}\!\!\longrightarrow \mathrm{Z} \odot \mathrm{Z} \longrightarrow \mathrm{C} \odot \mathrm{Z} \longrightarrow \mathrm{X}$
$parse(\pi, \{\}) = \{Y \to A; W \to B; Z \to C; fifo(X,Y); fifo(Y,W); fifo(W,Z)\}$

3.3 Single step relation go

After processing π with *parse*, the interpretation of the execution of π is given by $go(t, s, acc), go\colon s \times s \to s$, where s is a string denoting the processed program π as the one returned by *parse*, and t is the initial data flow of ports of the Reo program π. The parameter *acc* holds all connectors of the Reo circuit that satisfy their respective required conditions for data to flow.

For each interpretation π_i that composes π by *parse* in s, *go* is bound to verify whether π_i complies with the data described by t. If it does, then it is eligible to fire and *acc* holds its reference considering the nature of π_i. Definition 3.5 describes the semantics for the execution of a Reo program in *ReLo*. Due to lack of space, we are presenting only a fragment of the whole relation. Its full definition can be found in the project's repository.

Definition 3.5 [Relation *go* for a single execution]

- $s = \epsilon : fire(t, acc)$
- $s = A \to B \circ s'$:
  - · $go(t, s', acc \circ (A \to B))$, iff $Ax \prec t, (A \to B) \not\prec s'$
  - · $go(t, s', (acc \circ (A \to B)) \setminus s'_j) \cup go(t, s', acc)$, iff $Bx \prec t, (A \to B) \not\prec s'$ and $\exists s'_j \in acc \mid sink(s'_j) = B$
  - · $go(t, s', acc)$, otherwise
- $s = Sblock(A,B) \circ s'$:
  - · $go(t, s', acc)$, iff $(Ax \prec t \wedge Bx \prec t) \vee (Ax \not\prec t \wedge Bx \not\prec t)$ and $Sblock(A,B) \not\prec s'$
  - · $go(t, halt(A, B, s'), acc)$, iff $(Ax \prec t \wedge Bx \not\prec t) \vee (Ax \not\prec t \wedge Bx \prec t)$ and $Sblock(A,B) \not\prec s'$

The usage of *parse* is required to eliminate problems regarding the execution order of π's Reo channels, which could be caused by processing π as it is. Its interpretation is done by $go(t, s), go\colon s \times s \to s$, where s is a string containing π as processed by *parse*, and t is π's initial data arrangement. Condition $halt(A, B, s')$ removes every occurrence of firings that have A as its source node, whenever the required conditions for them to fire fails.

3.4 Data marking relation fire

As previously stated, *go* employs a function named $fire\colon s \times s \to s$, a function that returns the firing of all possible data flows in the Reo program, given the Reo program π and an initial data flow on ports of π. *ReLo*'s firing relation f is defined based on *go*'s definition as $f(t, \pi) = go(t, (g(\pi, [])), [])$, therefore relying on *fire*. Intuitively, *fire* controls whether the data flows. We define *fire* as follows:

$$fire(t,s) = \begin{cases} \epsilon, \text{ if } s = \epsilon \\ AxB \circ fire(t,s'), \text{ if } s = (AxB) \circ s' \text{ and } Ax \prec t \\ B(f(x)) \circ fire(t,s'), \text{ if } s = (f(a) \rightarrow B) \circ s' \text{ and } Ax \prec t \\ Bx \circ fire(t,s'), \text{ if } \begin{cases} s = (A \rightarrow B) \circ s' \text{ and } Ax \prec t, or \\ s = (AxB \rightarrow Bx) \circ s' \text{ and } AxB \prec t \end{cases} \end{cases} \tag{2}$$

We define f as the transition relation of a model. It denotes how the transitions of the model fire. Given an input t and a program π denoting a Reo circuit, $f(t, \pi)$ interfaces with go in order to return the resulting data flow of π.

$$f(t,\pi) = go(t, (parse(\pi, \{\})), \{\}) \tag{3}$$

3.5 Semantic notion of ReLo

We define *ReLo*'s semantic notion inductively as follows. Let $\mathcal{M} = \langle \mathcal{F}, \mathbf{V} \rangle$ and p, p_1 and p_2 be propositional formula. The notion of satisfaction of a formula p in $\mathcal{M}$ at a state $s \in S$ of the model, denoted by $\mathcal{M}, s \Vdash p$ is defined as follows. We present only a fragment of the semantics as the remainder of it is abbreviations of the presented rules.

- $\mathcal{M}$,$s \Vdash p$ iff $p \in V(s)$
- $\mathcal{M}$,$s \Vdash \top$ always
- $\mathcal{M}$,$s \Vdash \neg p$ iff $\mathcal{M}, s \nVdash p$
- $\mathcal{M}$,$s \Vdash p_1 \wedge p_2$ iff $\mathcal{M}, s \Vdash p_1$ and $\mathcal{M}, s \Vdash p_2$
- $\mathcal{M}$,$s \Vdash \langle t, \pi \rangle p$ if there is a state $w \in S$, $sR_\pi w$ and $p \in V(w)$. Intuitively, $\mathcal{M}, s \Vdash \langle t, \pi \rangle p$ holds if there is a state w reached from s by means of R_π (after execution of π with input t starting on state s) where p holds.

We recover the circuit in Fig. 2 to illustrate. Let us consider s = D_X, (i.e. t = D1) and the Sequencer's corresponding model $\mathcal{M}$. Therefore, $\mathcal{M}, D_X \Vdash \langle t, \pi \rangle p$ holds if $p \in V(D_{XfifoY})$ as D_{XfifoY} is the only state where $D_X R_\Pi D_{XfifoY}$. For example, one might state p as "There is no port with any data flow", hence $p \in V(D_{XfifoY})$.

3.6 Axiomatic System

We also define an axiomatization of *ReLo* based on other dynamic logics tailored to reason about programs. We discuss and define *ReLo*'s axioms and rules as follows. Let φ and ψ be formulas. Definition 3.6 introduces *ReLo*'s axiomatic system. We discuss the proofs of validity (w.r.t. *ReLo*'s model) of **(R)**, **(It)**, **(In)** in a file in the project's repository. Due to lack of space, we will not show them.

Definition 3.6 [Axiomatic System]

(PL) Enough Propositional Logic tautologies
(K) $[t,\pi](\varphi \rightarrow \psi) \rightarrow ([t,\pi]\varphi \rightarrow [t,\pi]\psi)$
(And) $[t,\pi](\varphi \wedge \psi) \leftrightarrow [t,\pi]\varphi \wedge [t,\pi]\varphi$
(Du) $[t,\pi]\varphi \leftrightarrow \neg\langle t,\pi\rangle\neg\varphi$
(R) $\langle t,\pi\rangle\varphi \leftrightarrow \varphi$ if $f(t,\pi) = \epsilon$
(It) $\varphi \wedge [t,\pi][t,\pi^\star]\varphi \leftrightarrow [t,\pi^\star]\varphi$
(In) $\varphi \wedge [t,\pi^\star](\varphi \rightarrow [t,\pi]\varphi) \rightarrow [t,\pi^\star]\varphi$

(MP) $\dfrac{\varphi \qquad \varphi \rightarrow \psi}{\psi}$

(Gen) $\dfrac{\varphi}{[t,\pi]\varphi}$

Axioms **(PL)**, **(K)**, **(And)** and **(Du)** are standard in Modal Logic literature, along with rules **(MP)** and **(Gen)** [7]. Axiom **(It)** respectively denotes the reasoning over nondeterministic iteration denoted by the operator *, following a similar idea portrayed by its counterpart for PDL. An intuitive interpretation of **(It)** is as follows. If φ holds in the current state and after a single execution of π with t, any finite nondeterministic number of iterations of π with t preserves φ's truth value, then φ must hold after any (nondeterministic finite) number of iterations of π with t.

Axiom **(In)** is an axiom which carries a similar idea as the one presented by [7] for Propositional Dynamic Logic. It enables the inductive reasoning on programs by carrying the following intuitive meaning: "considering that φ holds in the current state, if after any (nondeterministic finite) number of iterations of π with its respective input t φ holds, then it will hold after any number of iterations of π taking t as its input.

4 Usage Examples

As a usage Example, we recover the example from Fig. 2 and formalize some properties which may be interesting for this connector to have. Let us consider that the data markup is $t = X1$, $\mathcal{M}$ the model regarding the Sequencer, and the states' subscript denoting which part of the connector has data. The following lemma states that for this data flow, after every single execution of π, it is not the case that the three connected entities have their data equal to 1 simultaneously, but it does have data in its buffer from X to Y.

Example 4.1 $[X1, \pi]\neg(D_A = 1 \wedge D_B = 1 \wedge D_C = 1) \wedge t' = X1Y$, where $t' = f(t, \pi)$
$\mathcal{M}, D_X \Vdash [X1, \pi]\neg(D_A = 1 \wedge D_B = 1 \wedge D_C = 1) \wedge t' = X1Y$.
$\mathcal{M}, D_{X \rightarrow\boxed{\,}\rightarrow Y} \Vdash \neg(D_A = 1 \wedge D_B = 1 \wedge D_C = 1) \wedge t' = X1Y$.
$\mathcal{M}, D_{X \rightarrow\boxed{\,}\rightarrow Y} \Vdash \neg(D_A = 1 \wedge D_B = 1 \wedge D_C = 1)$ and $\mathcal{M}, D_{X \rightarrow\boxed{\,}\rightarrow Y} \Vdash t' = X1Y$.

It is also possible to verify that, if there is an execution of π that lasts a nondeterministic finite number of iterations, and there is data in C equal to 1, then there is an execution under the same circumstances where the same data is in B.

Example 4.2 $\langle X1, \pi^\star\rangle D_C = 1 \rightarrow \langle X1, \pi^\star\rangle D_B = 1$
$\mathcal{M}, D_X \Vdash \langle X1, \pi^\star\rangle D_C = 1 \rightarrow \langle X1, \pi^\star\rangle D_B = 1$
$\mathcal{M}, D_X \Vdash \neg(\langle X1, \pi^\star\rangle D_C = 1) \vee \langle X1, \pi^\star\rangle D_B = 1$
$\mathcal{M}, D_X \Vdash [X1, \pi^\star]\neg(D_C = 1) \vee \langle X1, \pi^\star\rangle D_B = 1$
$\mathcal{M}, D_X \Vdash [X1, \pi^\star]\neg(D_C = 1)$ or $\mathcal{M}, D_X \Vdash \langle X1, \pi^\star\rangle D_B = 1$
$\mathcal{M}, D_X \Vdash \langle X1, \pi^\star\rangle D_B = 1$, because $\mathcal{M}, D_B \Vdash D_B = 1$ and $D_X R_{\pi^\star} R_B$.

5 Conclusions and ongoing Work

Software engineers often employ techniques that reuse existing software components and/or use models as a starting point to develop a system. These approaches may bring a number of advantages, such as cost reduction and the mitigation of problems which would be later captured only in the testing phase (or not captured at all). For systems that require guarantees of some properties. The formal verification of models can be a valuable approach.

The present work introduces *ReLo*, a dynamic logic that offers a framework to naturally model and reason over Reo connectors. *ReLo*'s objective is to provide a formal system to naturally model and reason over Reo models and its properties (as the logic denotes Reo's semantics), not needing any translation mechanism to do so.

We are currently working on a *ReLo* implementation on Coq proof assistant to reason over Reo models. This implementation aims to bring some advantages, such as providing a computerized environment to reason over Reo models and to extract code regarding the connector's model, a communication pattern established by Reo of how the connected entities may interact to obtain the required system, ensuring specified requirements.

References

[1] Arbab, F., *Reo: a channel-based coordination model for component composition*, Mathematical Structures in Computer Science **14** (2004), p. 329–366.

[2] Arbab, F., *Coordination for component composition*, Electronic Notes in Theoretical Computer Science **160** (2006), pp. 15 – 40, proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005).

[3] Arbab, F., N. Kokash and S. Meng, *Towards using reo for compliance-aware business process modeling*, in: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, Springer, 2008, pp. 108–123.

[4] Atkinson, C. and T. Kuhne, *Model-driven development: a metamodeling foundation*, IEEE software **20** (2003), pp. 36–41.

[5] Baier, C., M. Sirjani, F. Arbab and J. Rutten, *Modeling component connectors in reo by constraint automata*, Science of computer programming **61** (2006), pp. 75–113.

[6] Emig, C., K. Krutz, S. Link, C. Momm and S. Abeck, *Model-driven development of soa services*, Universität Karlsruhe (TH), Karlsruhe (2007).

[7] Harel, D., D. Kozen and J. Tiuryn, *Dynamic logic*, in: *Handbook of philosophical logic*, Springer, 2001 pp. 99–217.

[8] Jongmans, S.-S. T. and F. Arbab, *Overview of thirty semantic formalisms for reo.*, Scientific Annals of Computer Science **22** (2012).

[9] Knight, J. C., *Safety critical systems: challenges and directions*, in: *Proceedings of the 24th International Conference on Software Engineering*, ACM, 2002, pp. 547–550.

[10] Kokash, N. and F. Arbab, "Formal Behavioral Modeling and Compliance Analysis for Service-Oriented Systems," Springer Berlin Heidelberg, Berlin, Heidelberg, 2009 pp. 21–41.

[11] Kokash, N., C. Krause and E. De Vink, *Reo+ mcrl2: A framework for model-checking dataflow in service compositions*, Formal Aspects of Computing **24** (2012), pp. 187–216.

[12] Lazovik, A. and F. Arbab, *Using reo for service coordination*, in: *International Conference on Service-Oriented Computing*, Springer, 2007, pp. 398–403.

[13] Ostro, J. S., *Formal methods for the specification and design of real-time safety critical systems*, Journal of Systems and Software **18** (1992), pp. 33–60.

[14] Papazoglou, M. P., *Service-oriented computing: Concepts, characteristics and directions*, in: *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, IEEE, 2003, pp. 3–12.

Pure Pattern Calculus *à la* de Bruijn[1]

Alexis Martín

Universidad de Buenos Aires, Argentina

Alejandro Ríos

Universidad de Buenos Aires, Argentina

Andrés Viso

Universidad de Buenos Aires, Argentina
Universidad Nacional de Quilmes, Argentina

Abstract

It is well-known in the field of programming languages that dealing with variable names and binders may lead to conflicts such as undesired captures when implementing interpreters or compilers. This situation has been overcome by resorting to de Bruijn indices for calculi where binders capture only one variable name, like the λ-calculus. The advantage of this approach relies on the fact that so-called α-equivalence becomes syntactical equality when working with indices.
In recent years pattern calculi have gained considerable attention given their expressiveness. They turn out to be notoriously convenient to study the foundations of modern functional programming languages modeling features like pattern matching, path polymorphism, pattern polymorphism, etc. However, the literature falls short when it comes to dealing with α-conversion and binders capturing simultaneously several variable names. Such is the case of the *Pure Pattern Calculus* (PPC): a natural extension of λ-calculus that allows to abstract virtually any term.
This paper extends de Bruijn's ideas to properly overcome the multi-binding problem by introducing a novel presentation of PPC with bidimensional indices, in an effort to implement a prototype for a typed functional programming language based on PPC that captures path polymorphism.

Keywords: de Bruijn indices, pattern calculi, pattern matching, α-equivalence.

1 Introduction

The foundations of functional programming languages like LISP, Miranda, Haskell or the ones in the ML family (Caml, SML, OCaml, etc.) rely strongly on the study of the λ-calculus [2] and its many variants introduced over the years. Among them there are the *pattern calculi* [23,7,16,6,14,12,18], whose key feature can be identified as *pattern-matching*. Pattern-matching has been extensively used in programming languages as a means for writing succinct and elegant programs. It stands for the possibility of defining functions by cases, analysing the shape of their arguments, while providing a syntactic tool to decompose such arguments in their parts when applying the function.

In the standard λ-calculus, functions are represented by expressions of the form $\lambda x.t$, where x is the formal parameter and t the body of the function. Such a function may be applied to any term, regardless of its form, as dictated by the β-reduction rule: $(\lambda x.t)\, u \mapsto_\beta \{x \setminus u\}t$, where $\{x \setminus u\}t$ stands for the result of replacing all free occurrences of x in t by u. Note that no requirement on the shape of u is placed. Pattern calculi, on the contrary, provide generalisations of the β-reduction rule in which abstractions $\lambda x.t$ are replaced by more

[1] This work was partially supported by LIA INFINIS, and the ECOS-Sud program PA17C01.

general terms like $\lambda p.t$ where p is called a *pattern*. For example, consider the function $\lambda\langle x, y\rangle.x$ that projects the first component of a pair. Here the pattern is the pair $\langle x, y\rangle$ and the expression $(\lambda\langle x, y\rangle.x)\,u$ will only be able to reduce if u is indeed of the form $\langle u_1, u_2\rangle$. Otherwise, reduction will be blocked.

We are particularly interested in studying the *Pure Pattern Calculus* (PPC) [15] and the novel features it introduced in the field of pattern calculi, namely *path polymorphism* and *pattern polymorphism*. The former refers to the possibility of defining functions that uniformly traverse arbitrary data structures, while the latter allows to consider patterns as parameters that may be dynamically generated in run-time. Developing such a calculus implies numerous technical challenges to guarantee well-behaved operational semantics in the untyped framework. Recently, a static type system has been introduced for a restriction of PPC called *Calculus of Applicative Patterns* (CAP) [28], which is able to capture the path polymorphic aspect of PPC. Moreover, type-checking algorithms for such a formalism has also been studied [10], as a first step towards an implementation of a prototype for a typed functional programming language capturing such features. Following this line of research, studies on the definition of normalising strategies for PPC have been done as well [3,4]. Such results are ported to CAP by means of a simple embedding [27] where the static typing discipline gives further guarantees on the well-behaved semantics of terms.

Within this framework, the present work aims to throw some light on the implementation aspects of these formalisms. In particular, working modulo α-conversion [2] implies dealing with variable renaming during the implementation. Such an approach is known to be error-prone and computationally expensive. One way of getting rid of this problem in the λ-calculus setting is adopting de Bruijn notation [8,9], a technique that simply avoids working modulo α-conversion. To the best of our knowledge, no dynamic pattern calculi in the likes of PPC with de Bruijn indices has been formalised in the literature. However, there are some references worth mentioning. In [24] an alternative presentation of PPC is given in the framework of *Higher-Order Pattern Rewriting System* (HRS) [22,20], together with translations between the two systems. On the other hand, in [5] de Bruijn ideas had been extended to *Expression Reduction Systems* (ERS) [11] also providing formal translations from systems with names to systems with indices, and vice-versa. Moreover, the correspondence between HRS and ERS has already been established [25]. The composition of such translations might derive a higher order system *à la* de Bruijn capturing the features of PPC. However, this would result in a rather indirect solution to our problem where many technicalities still need to be sorted out.

We aim to formalise an intuitive variant of PPC with de Bruijn indices where known results for the original calculus, such as the existence of normalising strategies, may easily be ported and reused.

1.1 Contributions

This paper extends de Bruijn's ideas to handle binders that capture multiple symbols at once, by means of what we call *bidimensional indices*. These ideas are illustrated by introducing a novel presentation of PPC, without variable/matchable names, called $\mathsf{PPC}_{\mathsf{dB}}$. Moreover, binders in the new proposed calculus are capable of handling two kinds of indices, namely variable and matchable indices, as required by the PPC operational semantics.

Proper translations from PPC to $\mathsf{PPC}_{\mathsf{dB}}$ and back are introduced. This functions preserve the matching operation and, hence, the operational semantics of both calculi. Moreover, they turn out to be the inverse of each other. This leads to a crucial strong bisimulation result between the two calculi, which allows to import many known properties of PPC into $\mathsf{PPC}_{\mathsf{dB}}$, for instance confluence and the existence of normalising strategies.

1.2 Structure of the paper

We start by briefly introducing PPC and reminding the mechanism of de Bruijn indices for the λ-calculus in Sec. 2. The novel $\mathsf{PPC}_{\mathsf{dB}}$ is formalised in Sec. 3, followed by the introduction of the translations in Sec. 4. The strong bisimulation result is presented in Sec. 5 together with a discussion of different properties of $\mathsf{PPC}_{\mathsf{dB}}$ that follow from it. We conclude in Sec. 6 and discuss possible lines of future work. Some technical details and proofs are relegated to the extended report available online [19].

2 Preliminaries

This section introduces preliminary concepts that guide our development and will help the reader follow the new ideas presented in this work.

2.1 The Pure Pattern Calculus

We start by briefly introducing the *Pure Pattern Calculus* (PPC) [15], an extension of the λ-calculus where virtually any term can be abstracted. This gives place to two versatile forms of polymorphism that set the foundations for adding novel features to future functional programming languages: namely *path polymorphism* and *pattern polymorphism*. This work, however, focuses on implementation related aspects of PPC and will not delve deeper into these new forms of polymorphism. We refer the reader to [15,13] for an in-depth study of them.

Given an infinitely countable set of *symbols* $\mathbb{V}$ $(x, y, z, \dots)$, the sets of *terms* $\mathbb{T}_{\mathsf{PPC}}$ and *contexts* are given by the following grammar:

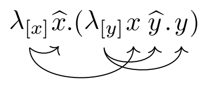
$$\textbf{Terms}\ t ::= x \mid \widehat{x} \mid t\,t \mid \lambda_\theta t.t \qquad \textbf{Contexts}\ \mathtt{C} ::= \Box \mid \mathtt{C}\,t \mid t\,\mathtt{C} \mid \lambda_\theta \mathtt{C}.t \mid \lambda_\theta t.\mathtt{C}$$

where θ is a list of symbols that are bound by the abstraction. A symbol x appearing in a term is dubbed a *variable symbol* while $\widehat{x}$ is called a *matchable symbol*. In particular, given $\lambda_\theta p.t$, θ binds variable symbols in the *body* t and matchable symbols in the *pattern* p. Thus, the set of *free variables* and *free matchables* of a term t, written $\mathsf{fv}(t)$ and $\mathsf{fm}(t)$ respectively, are inductively defined as:

$$\begin{aligned}
\mathsf{fv}(x) &\triangleq \{x\} & \mathsf{fm}(x) &\triangleq \emptyset \\
\mathsf{fv}(\widehat{x}) &\triangleq \emptyset & \mathsf{fm}(\widehat{x}) &\triangleq \{x\} \\
\mathsf{fv}(t\,u) &\triangleq \mathsf{fv}(t) \cup \mathsf{fv}(u) & \mathsf{fm}(t\,u) &\triangleq \mathsf{fm}(t) \cup \mathsf{fm}(u) \\
\mathsf{fv}(\lambda_\theta p.t) &\triangleq \mathsf{fv}(p) \cup (\mathsf{fv}(t) \setminus \theta) & \mathsf{fm}(\lambda_\theta p.t) &\triangleq (\mathsf{fm}(p) \setminus \theta) \cup \mathsf{fm}(t)
\end{aligned}$$

A term is said to be *closed* if it has no free variables. Note that free matchables are allowed, and should be understood as *constants* or *constructors* for data structures. The pattern p of an abstraction $\lambda_\theta p.t$ is *linear* if every symbol $x \in \theta$ occurs at most once in p.

To illustrate how variables and matchables are bound, consider the function elim defined as $\lambda_{[x]}\widehat{x}.(\lambda_{[y]} x\,\widehat{y}.y)$. The inner abstraction binds the only occurrence of the matchable $\widehat{y}$ in the pattern $x\,\widehat{y}$ and that of the variable y in the body y. However, the occurrence of x in $x\,\widehat{y}$ is not bound by the inner abstraction, as it is excluded from $[y]$, acting as a place-holder in that pattern. It is the outermost abstraction that binds both x in the inner pattern and $\widehat{x}$ in the outermost pattern. This is graphically depicted above.



A *substitution* $(\sigma, \rho, \dots)$ is a partial function from variables to terms. The substitution $\sigma = \{x_i \setminus u_i\}_{i \in I}$, where I is a set of indices, maps the variable x_i into the term u_i (*i.e.* $\sigma(x_i) \triangleq u_i$) for each $i \in I$. Thus, its *domain* and *image* are defined as $\mathsf{dom}(\sigma) \triangleq \{x_i\}_{i \in I}$ and $\mathsf{img}(\sigma) \triangleq \{u_i\}_{i \in I}$ respectively. For convenience, a substitution σ is usually turned into a total function by defining $\sigma(x) \triangleq x$ for every $x \notin \mathsf{dom}(\sigma)$. Then, the identity substitution is denoted $\{\}$ or simply *id*.

A *match* $(\mu, \nu, \dots)$ may be successful (yielding a substitution), it may fail (returning a special symbol `fail`) or be undetermined (denoted by a special symbol `wait`). The cases of success and failure are called *decided matches*. All concepts and notation relative to substitutions are extended to matches so that, for example, the domain of `fail` is empty while that of `wait` is undefined. The sets of free variable and free matchable symbols of σ are defined as the union of $\mathsf{fv}(\sigma x)$ and $\mathsf{fm}(\sigma x)$ for every $x \in \mathsf{dom}(\sigma)$ respectively, while $\mathsf{fv}(\mathtt{fail}) = \mathsf{fm}(\mathtt{fail}) = \emptyset$ and they are undefined for `wait`. The set of symbols of a substitution is defined as $\mathsf{sym}(\sigma) \triangleq \mathsf{dom}(\sigma) \cup \mathsf{fv}(\sigma) \cup \mathsf{fm}(\sigma)$. The predicate x `avoids` σ states that $x \notin \mathsf{sym}(\sigma)$. It is extended to sets and matches as expected. In particular, θ `avoids` μ implies that μ must be decided.

The result of applying a substitution σ to a term t, denoted σt, is inductively defined as:

$$\begin{aligned}
\sigma x &\triangleq \sigma(x) & \sigma(t\,u) &\triangleq \sigma t\,\sigma u \\
\sigma \widehat{x} &\triangleq \widehat{x} & \sigma \lambda_\theta p.t &\triangleq \lambda_\theta \sigma p.\sigma t \quad \text{if } \theta\ \mathtt{avoids}\ \sigma
\end{aligned}$$

The restriction in the case of the abstraction is required to avoid undesired captures of variables/matchables. However, it can always be satisfied by resorting to α-conversion.

The result of applying a match μ to a term t, denoted $\mu\,t$, is defined as: (i) if $\mu = \sigma$ a substitution, then $\mu\,t \triangleq \sigma t$; (ii) if $\mu = \mathtt{fail}$, then $\mu\,t \triangleq \lambda_{[x]}\widehat{x}.x$ (*i.e.* the identity function); or (iii) if $\mu = \mathtt{wait}$, then $\mu\,t$ is undefined.

The *composition* $\sigma \circ \sigma'$ of substitutions is defined as usual, *i.e.* $(\sigma \circ \sigma')x \triangleq \sigma(\sigma' x)$, and the notion is extended to matches by defining $\mu \circ \mu' \triangleq \mathtt{fail}$ if any of the two matches is `fail`. Otherwise, if at least one of the two is `wait`, then $\mu \circ \mu' \triangleq \mathtt{wait}$. In particular, $\mathtt{fail} \circ \mathtt{wait} = \mathtt{fail}$. The *disjoint union* $\mu \uplus \mu'$ of matches is defined as follows: (i) if $\mu = \mathtt{fail}$ or $\mu' = \mathtt{fail}$, then $\mu \uplus \mu' \triangleq \mathtt{fail}$; else (ii) if $\mu = \mathtt{wait}$ or $\mu' = \mathtt{wait}$, then $\mu \uplus \mu' \triangleq \mathtt{wait}$;

otherwise (iii) both μ and μ' are substitutions and if $\mathsf{dom}(\mu) \cap \mathsf{dom}(\mu') \neq \emptyset$, then $\mu \uplus \mu' \triangleq \mathtt{fail}$, else:

$$(\mu \uplus \mu')x \triangleq \begin{cases} \mu x & \text{if } x \in \mathsf{dom}(\mu) \\ \mu' x & \text{if } x \in \mathsf{dom}(\mu') \\ x & \text{otherwise} \end{cases}$$

Disjoint union is used to guarantee that the matching operation is deterministic.

Before introducing the matching operation it is necessary to motivate the concept of *matchable form*. The pattern $\widehat{x}\,\widehat{y}$ allows, at first, to decompose arbitrary applications, which may lead to the loss of confluence. For instance:

$$\begin{array}{l} (\lambda_{[x,y]}\widehat{x}\,\widehat{y}.y)\,((\lambda_{[w]}\widehat{w}.\widehat{z}_0\,\widehat{z}_1)\,\widehat{z}_0) \to (\lambda_{[x,y]}\widehat{x}\,\widehat{y}.y)\,(\widehat{z}_0\,\widehat{z}_1) \to \widehat{z}_1 \\ (\lambda_{[x,y]}\widehat{x}\,\widehat{y}.y)\,((\lambda_{[w]}\widehat{w}.\widehat{z}_0\,\widehat{z}_1)\,\widehat{z}_0) \to \widehat{z}_0 \end{array}$$

This issue arises when allowing to match the pattern $\widehat{x}\,\widehat{y}$ with an application that may still be reduced, like the argument $(\lambda_{[w]}\widehat{w}.\widehat{z}_0\,\widehat{z}_1)\,\widehat{z}_0$ of the outermost redex in the example above. To avoid this situation it is required for the match to be decided only if the argument is sufficiently evaluated. An analogous issue occurs if the pattern is reducible. Thus, both the pattern and the argument must be in matchable form for the match to be decided. The set of *data structures* $\mathbb{D}_{\mathsf{PPC}}$ and *matchable forms* $\mathbb{M}_{\mathsf{PPC}}$ are given by the following grammar:

Data structures $d ::= \widehat{x} \mid d\,t$ **Matchable forms** $m ::= d \mid \lambda_\theta t.t$

The *matching operation* $\{\!\{p \setminus^\theta u\}\!\}$ of a pattern p against a term u relative to a list of symbols θ is defined as the application, in order, of the following equations:

$$\begin{array}{rcll} \{\!\{\widehat{x} \setminus^\theta u\}\!\} & \triangleq & \{x \setminus u\} & \text{if } x \in \theta \\ \{\!\{\widehat{x} \setminus^\theta \widehat{x}\}\!\} & \triangleq & \{\} & \text{if } x \notin \theta \\ \{\!\{p\,q \setminus^\theta t\,u\}\!\} & \triangleq & \{\!\{p \setminus^\theta t\}\!\} \uplus \{\!\{q \setminus^\theta u\}\!\} & \text{if } t\,u, p\,q \in \mathbb{M}_{\mathsf{PPC}} \\ \{\!\{p \setminus^\theta u\}\!\} & \triangleq & \mathtt{fail} & \text{if } u, p \in \mathbb{M}_{\mathsf{PPC}} \\ \{\!\{p \setminus^\theta u\}\!\} & \triangleq & \mathtt{wait} & \text{otherwise} \end{array}$$

An additional check is imposed, namely $\mathsf{dom}(\{\!\{p \setminus^\theta u\}\!\}) = \theta$. Otherwise, $\{\!\{p \setminus^\theta u\}\!\} \triangleq \mathtt{fail}$. This last condition is necessary to prevent bound symbols from going out of scope when reducing. It can be easily guaranteed though by requesting, for each abstraction $\lambda_\theta p.t$, that $\theta \subseteq \mathsf{fm}(p)$. For instance, consider the term $(\lambda_{[x,y]}\widehat{x}.y)\,u$. Without this final check, matching the argument u against the pattern $\widehat{x}$ would yield a substitution $\{x \setminus u\}$ and no term would be assigned to the variable y in the body of the abstraction.

Finally, the reduction relation $\to_{\mathsf{PPC}}$ of PPC is given by the closure by contexts of the rewriting rule:

$$(\lambda_\theta p.s)\,u \mapsto_{\mathsf{PPC}} \{\!\{p \setminus^\theta u\}\!\}\,s$$

whenever $\{\!\{p \setminus^\theta u\}\!\}$ is a decided match. To illustrate the operational semantics of PPC consider the term elim introduced above, applied to the function $\lambda_{[z]}\widehat{z}.\widehat{c}\,z\,\widehat{n}$ where the free matchables $\widehat{c}$ and $\widehat{n}$ can be seen as constructors for lists cons and nil respectively:

$$(\lambda_{[x]}\widehat{x}.(\lambda_{[y]}x\,\widehat{y}.y))\,(\lambda_{[z]}\widehat{z}.\widehat{c}\,z\,\widehat{n}) \quad \to_{\mathsf{PPC}} \quad \lambda_{[y]}(\lambda_{[z]}\widehat{z}.\widehat{c}\,z\,\widehat{n})\,\widehat{y}.y \quad \to_{\mathsf{PPC}} \quad \lambda_{[y]}\widehat{c}\,\widehat{y}\,\widehat{n}.y$$

In the first step, $\lambda_{[z]}\widehat{z}.\widehat{c}\,z\,\widehat{n}$ is substituted for x into the pattern $x\,\widehat{y}$. In the second step, the resulting application, which resides in the pattern, is reduced. The resulting term, when applied to an argument, will yield a successful matching only if this argument is a compound data of the form $\widehat{c}\,t\,\widehat{n}$.

This relation is shown to be *confluent* (CR) based on the matching operation introduced above.

Theorem 2.1 ([15]) *The reduction relation $\to_{\mathsf{PPC}}$ is confluent (CR).*

2.2 de Bruijn indices

We introduce next de Bruijn indices for the λ-calculus. Among the many presentations of de Bruijn indices in the literature, we will follow that of [17] as our development builds upon their ideas. In particular, we choose to work with the presentation where indices are partially updated as the term is being traversed by the substitution operation (details below). We refer the reader to [17] for the equivalent version where the update is performed once at the end of the substitution process. We introduce now the *λ-calculus with de Bruijn indices* ($\lambda_{\mathtt{dB}}$ for short).

The sets of *terms* $\mathbb{T}_{\lambda_{\mathtt{dB}}}$ and *contexts* are given by the following grammar:

$$\textbf{Terms } t ::= \mathtt{i} \mid t\,t \mid \lambda t \qquad \textbf{Contexts } \mathtt{C} ::= \Box \mid \mathtt{C}\,t \mid t\,\mathtt{C} \mid \lambda\mathtt{C}$$

where $\mathtt{i} \in \mathbb{N}_{\geq 1}$ is called an *index*. Indices are place-holders indicating the distance to the binding abstraction. In the context of the $\lambda_{\mathtt{dB}}$-calculus, indices are also called *variables*. Thus, the *free variables* of a term are inductively defined as: $\mathsf{fv}(\mathtt{i}) \triangleq \{\mathtt{i}\}$; $\mathsf{fv}(t\,u) \triangleq \mathsf{fv}(t) \cup \mathsf{fv}(u)$; and $\mathsf{fv}(\lambda t) \triangleq \mathsf{fv}(t) - 1$, where $X - k$ stands for subtracting k from each element of the set X, removing those that result in a non-positive index.

In order to define β-reduction *à la* de Bruijn, the substitution of an index $\mathtt{i}$ for a term u in a term t must be defined. Therefore, it is necessary to identify among the indices of the term t, those corresponding to $\mathtt{i}$. Furthermore, the indices of u should be updated in order to preserve the correct bindings after the replacement of the variable by u. To that end, the *increment at depth k* for variables in a term t, written $\uparrow_k(t)$, is inductively defined as follows:

$$\uparrow_k(\mathtt{i}) \triangleq \begin{cases} \mathtt{i}+1 & \text{if } i > k \\ \mathtt{i} & \text{if } i \leq k \end{cases} \qquad \begin{array}{r} \uparrow_k(t\,u) \triangleq \uparrow_k(t)\uparrow_k(u) \\ \uparrow_k(\lambda t) \triangleq \lambda\uparrow_{k+1}(t) \end{array}$$

Then, the *substitution at level i* of a term u in a term t, denoted $\{\mathtt{i} \setminus u\}t$, is defined as a partial function mapping free variables at level i to terms, performing the appropriate updates as it traverses the substituted term, to avoid undesired captures.

$$\{\mathtt{i} \setminus u\}\mathtt{i}' \triangleq \begin{cases} \mathtt{i}'-1 & \text{if } i' > i \\ u & \text{if } i' = i \\ \mathtt{i}' & \text{if } i' < i \end{cases} \qquad \begin{array}{r} \{\mathtt{i} \setminus u\}(t\,s) \triangleq \{\mathtt{i} \setminus u\}t\,\{\mathtt{i} \setminus u\}s \\ \{\mathtt{i} \setminus u\}\lambda t \triangleq \lambda\{\mathtt{i}+1 \setminus \uparrow_0(u)\}t \end{array}$$

It is worth noticing that this substitution should be interpreted in the context of a redex, where a binder is removed and its bound index substituted. This forces to update the free indices, that might be captured by an outermost abstraction, as done by the first case of the substitution over a variable $\mathtt{i}'$. Hence, preserving the correct bindings.

Finally, the reduction relation $\rightarrow_{\mathtt{dB}}$ of the $\lambda_{\mathtt{dB}}$-calculus is given by the closure by contexts of the rewriting rule:

$$(\lambda s)\,u \mapsto_{\mathtt{dB}} \{\mathtt{1} \setminus u\}s$$

Also, embeddings between the λ-calculus and $\lambda_{\mathtt{dB}}$ are defined: $[\![\_]\!] : \mathbb{T}_\lambda \to \mathbb{T}_{\lambda_{\mathtt{dB}}}$ and $(\!|\_|\!) : \mathbb{T}_{\lambda_{\mathtt{dB}}} \to \mathbb{T}_\lambda$, in such a way that they are the inverse of each other and, they allow to simulate one calculus into the other:

Theorem 2.2 **([17])** *Let $t \in \mathbb{T}_\lambda$ and $s \in \mathbb{T}_{\lambda_{\mathtt{dB}}}$. Then,*

(i) *If $t \rightarrow_\beta t'$, then $[\![t]\!] \rightarrow_{\mathtt{dB}} [\![t']\!]$.*

(ii) *If $s \rightarrow_{\mathtt{dB}} s'$, then $(\!|s|\!) \rightarrow_\beta (\!|s'|\!)$.*

This shows that both formalisms (λ-calculus and $\lambda_{\mathtt{dB}}$) have exactly the same operational semantics.

As an example to illustrate both reduction in the $\lambda_{\mathtt{dB}}$-calculus and its equivalence with the λ-calculus, consider the following terms: $(\lambda z.\lambda y.z)\,(\lambda x.x)\,(\lambda x.x\,x)$ and $(\lambda\lambda\mathtt{2})\,(\lambda\mathtt{1})\,(\lambda\mathtt{1}\,\mathtt{1})$. The reader can verify that both expressions encode the same function in its respective calculus. As expected, their operational semantics coincide

$$\begin{array}{ccccc} (\lambda z.\lambda y.z)\,(\lambda x.x)\,(\lambda x.x\,x) & \rightarrow_\beta & (\lambda y.\lambda x.x)\,(\lambda x.x\,x) & \rightarrow_\beta & \lambda x.x \\ (\lambda\lambda\mathtt{2})\,(\lambda\mathtt{1})\,(\lambda\mathtt{1}\,\mathtt{1}) & \rightarrow_{\mathtt{dB}} & (\lambda\lambda\mathtt{1})\,(\lambda\mathtt{1}\,\mathtt{1}) & \rightarrow_{\mathtt{dB}} & \lambda\mathtt{1} \end{array}$$

3 The Pure Pattern Calculus with de Bruijn indices

This section introduces the novel *Pure Pattern Calculus with de Bruijn indices* ($\mathsf{PPC}_{\mathtt{dB}}$). It represents a natural extension of de Bruijn ideas to a framework where a binder may capture more than one symbol. In the particular case of PPC there are two kinds of captured symbols, namely variables and matchables. This distinction is preserved in $\mathsf{PPC_{dB}}$ while extending indices to pairs (*a.k.a.* bidimensional indices) to distinguish the binder that captures the symbol and the individual symbol among all those captured by the same binder.

The sets of *terms* $\mathbb{T}_{\mathsf{PPC_{dB}}}$, *contexts*, *data structures* $\mathbb{D}_{\mathsf{PPC_{dB}}}$ and *matchable forms* $\mathbb{M}_{\mathsf{PPC_{dB}}}$ of $\mathsf{PPC_{dB}}$ are given by the following grammar:

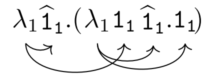
$$\begin{array}{rlrl}
\textbf{Terms} & t ::= \mathtt{i}_\mathtt{j} \mid \widehat{\mathtt{i}}_\mathtt{j} \mid t\,t \mid \lambda_n t.t & \textbf{Data structures} & d ::= \widehat{\mathtt{i}}_\mathtt{j} \mid d\,t \\
\textbf{Contexts} & \mathtt{C} ::= \Box \mid \mathtt{C}\,t \mid t\,\mathtt{C} \mid \lambda_n \mathtt{C}.t \mid \lambda_n t.\mathtt{C} & \textbf{Matchable forms} & m ::= d \mid \lambda_n t.t
\end{array}$$

where $\mathtt{i}_\mathtt{j}$ is dubbed a *bidimensional index* and denotes an ordered pair in $\mathbb{N}_{\geq 1} \times \mathbb{N}_{\geq 1}$ with *primary index* $\mathtt{i}$ and *secondary index* $\mathtt{j}$. The sub-index $n \in \mathbb{N}$ in an abstraction represents the amount of indices (pairs) being captured by it. The primary index of a pair is used to determine if the pair is bound by an abstraction, while the secondary index identifies the pair among those (possibly many) bound ones. As for PPC, an index of the form $\mathtt{i}_\mathtt{j}$ is called a *variable index* while $\widehat{\mathtt{i}}_\mathtt{j}$ is dubbed a *matchable index*. The *free variables* and *free matchables* of a term are thus defined as follows:

$$\begin{array}{rclrcl}
\mathsf{fv}(\mathtt{i}_\mathtt{j}) & \triangleq & \{\mathtt{i}_\mathtt{j}\} & \mathsf{fm}(\mathtt{i}_\mathtt{j}) & \triangleq & \emptyset \\
\mathsf{fv}(\widehat{\mathtt{i}}_\mathtt{j}) & \triangleq & \emptyset & \mathsf{fm}(\widehat{\mathtt{i}}_\mathtt{j}) & \triangleq & \{\mathtt{i}_\mathtt{j}\} \\
\mathsf{fv}(t\,u) & \triangleq & \mathsf{fv}(t) \cup \mathsf{fv}(u) & \mathsf{fm}(t\,u) & \triangleq & \mathsf{fm}(t) \cup \mathsf{fm}(u) \\
\mathsf{fv}(\lambda_n p.t) & \triangleq & \mathsf{fv}(p) \cup (\mathsf{fv}(t) - 1) & \mathsf{fm}(\lambda_n p.t) & \triangleq & (\mathsf{fm}(p) - 1) \cup \mathsf{fm}(t)
\end{array}$$

where $X - k$ stands for subtracting k from the primary index of each element of the set X, removing those that result in a non-positive index.

Let us illustrate these concepts with a similar example as that given for PPC, namely the function $\mathsf{elim} = \lambda_{[x]}\widehat{x}.(\lambda_{[y]} x\,\widehat{y}.y)$. An equivalent term in the $\mathsf{PPC_{dB}}$ framework would be $\lambda_1 \widehat{\mathtt{1}}_\mathtt{1}.(\lambda_1 \mathtt{1}_\mathtt{1}\,\widehat{\mathtt{1}}_\mathtt{1}.\mathtt{1}_\mathtt{1})$. Note that variable indices in the context of a pattern are not bound by the respective abstraction, in the same way that matchable indices in the body of the abstraction are not captured either. Thus, the first occurrence of $\mathtt{1}_\mathtt{1}$ is actually bound by the outermost abstraction, together with the first occurrence of the matchable index $\widehat{\mathtt{1}}_\mathtt{1}$. The rest of the indices in the term are bound by the inner abstraction as depicted in the figure to the right. As a further (more interesting) example, consider the term $\lambda_{[x,y]}\widehat{x}\,\widehat{y}.\lambda_{[]}x.y$ from PPC, whose counter-part in $\mathsf{PPC_{dB}}$ would look like $\lambda_2 \widehat{\mathtt{1}}_\mathtt{1}\,\widehat{\mathtt{1}}_\mathtt{2}.\lambda_0 \mathtt{1}_\mathtt{1}.\mathtt{2}_\mathtt{2}$. This example illustrates the use of secondary indices to identify symbols bound by the same abstraction. It also shows how the primary index of a variable is increased when occurring within the body of an internal abstraction, while this is not the case for occurrences in a pattern position. Thus, both $\mathtt{1}_\mathtt{1}$ and $\mathtt{2}_\mathtt{2}$ are bound by the outermost abstraction, as well as $\widehat{\mathtt{1}}_\mathtt{1}$ and $\widehat{\mathtt{1}}_\mathtt{2}$. Note how the inner abstraction does not bind any index at all.



A term t is said to be *well-formed* if all the free bidimensional indices (variables and matchables) of t have their secondary index equal to 1, and for every sub-term of the form $\lambda_n p.s$ (written $\lambda_n p.s \subseteq t$) all the pairs captured by the abstraction have their secondary index within the range $[1, n]$. Formally, $\{\mathtt{i}_\mathtt{j} \mid \mathtt{i}_\mathtt{j} \in \mathsf{fm}(t) \cup \mathsf{fv}(t), j > 1\} \cup (\bigcup_{\lambda_n p.s \subseteq t} \{\mathtt{1}_\mathtt{j} \mid \mathtt{1}_\mathtt{j} \in \mathsf{fm}(p) \cup \mathsf{fv}(s), j > n\}) = \emptyset$.

Before introducing a proper notion of substitution for $\mathsf{PPC_{dB}}$ it is necessary to have a mechanism to update indices at a certain depth within the term. The *increment at depth* k for variable and matchable indices in a term t, written $\uparrow_k(t)$ and $\Uparrow_k(t)$ respectively, are inductively defined as follows

$$\begin{array}{rclrcl}
\uparrow_k(\mathtt{i}_\mathtt{j}) & \triangleq & \begin{cases} (\mathtt{i}+\mathtt{1})_\mathtt{j} & \text{if } i > k \\ \mathtt{i}_\mathtt{j} & \text{if } i \leq k \end{cases} & \Uparrow_k(\mathtt{i}_\mathtt{j}) & \triangleq & \mathtt{i}_\mathtt{j} \\
\uparrow_k(\widehat{\mathtt{i}}_\mathtt{j}) & \triangleq & \widehat{\mathtt{i}}_\mathtt{j} & \Uparrow_k(\widehat{\mathtt{i}}_\mathtt{j}) & \triangleq & \begin{cases} \widehat{(\mathtt{i}+\mathtt{1})}_\mathtt{j} & \text{if } i > k \\ \widehat{\mathtt{i}}_\mathtt{j} & \text{if } i \leq k \end{cases} \\
\uparrow_k(t\,u) & \triangleq & \uparrow_k(t)\uparrow_k(u) & \Uparrow_k(t\,u) & \triangleq & \Uparrow_k(t)\Uparrow_k(u) \\
\uparrow_k(\lambda_n p.t) & \triangleq & \lambda_n \uparrow_k(p).\uparrow_{k+1}(t) & \Uparrow_k(\lambda_n p.t) & \triangleq & \lambda_n \Uparrow_{k+1}(p).\Uparrow_k(t)
\end{array}$$

Similarly, the *decrement at depth* k for variables ($\downarrow_k(\_)$) and matchables ($\Downarrow_k(\_)$) are defined by subtracting one from the primary index above k in the term. Most of the times these functions are used with $k = 0$, thus the subindex will be omitted when it is clear from context. In particular, the decrement function for variables will allow us to generalise the idea of substitution at level i with respect to the original one presented in Sec. 2.2, which only holds in the context of a β-reduction, by making the necessary adjustments to the indices at the moment of the redution instead of hard-coding them into the substitution meta-operation.

A *substitution at level* i is a partial function from variable indices to terms. It maps free variable indices at level i to terms, performing the appropriate updates as it traverses the substituted term, to avoid undesired captures.

$$\{\mathtt{i}_\mathtt{j} \setminus u_j\}_{j\in J}\mathtt{i'}_\mathtt{k} \triangleq \begin{cases} u_k & \text{if } i' = i, k \in J \\ \mathtt{i'}_\mathtt{k} & \text{if } i' \neq i \end{cases} \qquad \{\mathtt{i}_\mathtt{j} \setminus u_j\}_{j\in J}(t\,s) \triangleq \{\mathtt{i}_\mathtt{j} \setminus u_j\}_{j\in J}t\,\{\mathtt{i}_\mathtt{j} \setminus u_j\}_{j\in J}s$$

$$\{\mathtt{i}_\mathtt{j} \setminus u_j\}_{j\in J}\widehat{\mathtt{i'}}_\mathtt{k} \triangleq \widehat{\mathtt{i'}}_\mathtt{k} \qquad \{\mathtt{i}_\mathtt{j} \setminus u_j\}_{j\in J}\lambda_n p.t \triangleq \lambda_n\{\mathtt{i}_\mathtt{j} \setminus {\Uparrow}(u_j)\}_{j\in J}p.\{(\mathtt{i}+\mathtt{1})_\mathtt{j} \setminus {\uparrow}(u_j)\}_{j\in J}t$$

It is worth noticing that the base case for variable indices is undefined if $i' = i$ and $k \notin J$. Such case will render the result of the substitution undefined as well. In the operational semantics of $\mathsf{PPC}_{\mathtt{dB}}$, the matching operation presented below will be responsible for avoiding this undesired situation, as we will see later. The *domain* of a substitution at level i is given by $\mathsf{dom}(\{\mathtt{i}_\mathtt{j} \setminus u_j\}_{j\in J}) \triangleq \{\mathtt{i}_\mathtt{j}\}_{j\in J}$. The identity substitution (*i.e.* with empty domain) is denoted $\{\}$ or id.

As in PPC, a match $(\mu, \nu, \ldots)$ may succeed, fail ($\mathtt{fail}$) or be undetermined ($\mathtt{wait}$). For $\mathsf{PPC}_{\mathtt{dB}}$, a successful match will yield a substitution at level 1, as given by the following *matching operation*, where the rules are applied in order as in PPC:

$$\begin{array}{rcll}
\{\!\!\{\widehat{\mathtt{1}}_\mathtt{j} \setminus^n u\}\!\!\} & \triangleq & \{\mathtt{1}_\mathtt{j} \setminus u\} & \\
\{\!\!\{\widehat{\mathtt{i}+\mathtt{1}}_\mathtt{j} \setminus^n \widehat{\mathtt{i}}_\mathtt{j}\}\!\!\} & \triangleq & \{\} & \\
\{\!\!\{p\,q \setminus^n t\,u\}\!\!\} & \triangleq & \{\!\!\{p \setminus^n t\}\!\!\} \uplus \{\!\!\{q \setminus^n u\}\!\!\} & \text{if } t\,u, p\,q \in \mathbb{M}_{\mathsf{PPC}_{\mathtt{dB}}} \\
\{\!\!\{p \setminus^n u\}\!\!\} & \triangleq & \mathtt{fail} & \text{if } u, p \in \mathbb{M}_{\mathsf{PPC}_{\mathtt{dB}}} \\
\{\!\!\{p \setminus^n u\}\!\!\} & \triangleq & \mathtt{wait} & \text{otherwise}
\end{array}$$

where disjoint union of matching is adapted to $\mathsf{PPC}_{\mathtt{dB}}$ from PPC in a straightforward way. The first two rules in the matching operation for $\mathsf{PPC}_{\mathtt{dB}}$ are worth a comment. As the matching operation should be understood in the context of a redex, the matchable symbols bound in the pattern are those with primary index equal to 1. Thus, $\mathsf{PPC}_{\mathtt{dB}}$'s counter-part of the membership check $x \in \theta$ from PPC's matching operation is a simple syntactic check on the primary index. Similarly, $x \notin \theta$ corresponds to the primary index being greater than 1, as checked by the second rule of the definition. However, a primary index $\widehat{\mathtt{i}+\mathtt{1}}$ within the pattern should match primary index $\widehat{\mathtt{i}}$ from the argument, since the former is affected by an extra binder in a redex. For instance, in the term $(\lambda_1\widehat{\mathtt{2}}_1\,\widehat{\mathtt{1}}_1.\mathtt{1}_1)\,(\widehat{\mathtt{1}}_1\,t')$ the matchable $\widehat{\mathtt{2}}_1$ is free and corresponds to $\widehat{\mathtt{1}}_1$ in the argument, while $\widehat{\mathtt{1}}_1$ from the pattern is bound by the abstraction. Its counter-part in PPC would be α-equivalent to $(\lambda_{[x]}\widehat{y}\,\widehat{x}.x)\,(\widehat{y}\,t)$. Hence, $\{\!\!\{\widehat{\mathtt{2}}_1\,\widehat{\mathtt{1}}_1 \setminus^1 \widehat{\mathtt{1}}_1\,t'\}\!\!\} = \{\mathtt{1}_1 \setminus t'\}$.

As for PPC, an additional post-condition is checked over $\{\!\!\{p \setminus^n u\}\!\!\}$ to prevent indices from going out of scope. It requires $\mathsf{dom}(\{\!\!\{p \setminus^n u\}\!\!\}) = \{\mathtt{1}_1, \ldots, \mathtt{1}_\mathtt{n}\}$, which essentially implies that all the bound indices are assigned a value by the resulting substitution. This condition can be guaranteed by requesting $\{\mathtt{1}_1, \ldots, \mathtt{1}_\mathtt{n}\} \subseteq \mathsf{fm}(p)$, for each abstraction $\lambda_n p.t$ within a well-formed term. To illustrate the need of such a check, consider the term $(\lambda_2\widehat{\mathtt{1}}_1.\mathtt{1}_2)\,u'$ (*i.e.* the $\mathsf{PPC}_{\mathtt{dB}}$ counter-part of $(\lambda_{[x,y]}\widehat{x}.y)\,u$, given in Sec. 2.1). If the matching $\{\!\!\{\widehat{\mathtt{1}}_1 \setminus^2 u'\}\!\!\} = \{\mathtt{1}_1 \setminus u'\}$ is considered correct, then no replacement for the variable index $\mathtt{1}_2$ in the body of the abstraction is set, resulting in an ill-behaved operational semantics.

The reduction relation $\rightarrow_{\mathtt{dB}}$ of $\mathsf{PPC}_{\mathtt{dB}}$ is given by the closure by contexts of the rewriting rule:

$$(\lambda_n p.s)\,u \mapsto_{\mathtt{dB}} {\downarrow}(\{\!\!\{p \setminus^n {\uparrow}(u)\}\!\!\}\,s)$$

whenever $\{\!\!\{p \setminus^n {\uparrow}(u)\}\!\!\}$ is a decided match. The decrement function for variable indices is applied to the reduct to compensate for the loss of a binder over s. However, the variable indices of u are not affected by such binder in the redex. Hence the need of incrementing them prior to the (eventual) substitution.

Following the reduction example given above for PPC, consider these codifications of $\mathsf{elim} = \lambda_{[x]}\widehat{x}.(\lambda_{[y]}x\,\widehat{y}.y)$ and $\lambda_{[z]}\widehat{z}.\widehat{c}\,z\,\widehat{n}$ respectively: $\lambda_1\widehat{\mathtt{1}}_1.(\lambda_1\mathtt{1}_1\,\widehat{\mathtt{1}}_1.\mathtt{1}_1)$ and $\lambda_1\widehat{\mathtt{1}}_1.\widehat{\mathtt{1}}_1\,\mathtt{1}_1\,\widehat{\mathtt{2}}_1$. Note how the first occurrence of $\mathtt{1}_1$ is actually bound by the outermost abstraction, since abstractions do not bind variable indices in their pattern. Similarly, the matchable index $\widehat{\mathtt{1}}_1$ in the body of $\lambda_1\widehat{\mathtt{1}}_1.\widehat{\mathtt{1}}_1\,\mathtt{1}_1\,\widehat{\mathtt{2}}_1$ turns out to be free as well as $\widehat{\mathtt{2}}_1$. Then, as expected, we have the following sequence:

$$(\lambda_1 \widehat{\mathtt{1}}_1.(\lambda_1 \mathtt{1}_1\, \widehat{\mathtt{1}}_1.\mathtt{1}_1))\,(\lambda_1 \widehat{\mathtt{1}}_1.\widehat{\mathtt{1}}_1\, \mathtt{1}_1\, \widehat{\mathtt{2}}_1) \quad \rightarrow_{\mathtt{dB}} \quad \lambda_1(\lambda_1 \widehat{\mathtt{1}}_1.\widehat{\mathtt{2}}_1\, \mathtt{1}_1\, \widehat{\mathtt{3}}_1)\, \widehat{\mathtt{1}}_1.\mathtt{1}_1 \quad \rightarrow_{\mathtt{dB}} \quad \lambda_1 \widehat{\mathtt{2}}_1\, \widehat{\mathtt{1}}_1\, \widehat{\mathtt{3}}_1.\mathtt{1}_1$$

In the first step, $\lambda_1 \widehat{\mathtt{1}}_1.\widehat{\mathtt{1}}_1\, \mathtt{1}_1\, \widehat{\mathtt{2}}_1$ is substituted for $\mathtt{1}_1$ into the pattern $\mathtt{1}_1\, \widehat{\mathtt{1}}_1$. The fact that the substitution takes place within the context of a pattern forces the application of $\Uparrow(\_)$, thus updating the matchable indices and obtaining $\lambda_1 \widehat{\mathtt{1}}_1.\widehat{\mathtt{2}}_1\, \mathtt{1}_1\, \widehat{\mathtt{3}}_1$. Note that the increment and decrement added by the reduccion rule take no effect as there are no free variable indices in the term. In the second step, the resulting application is reduced, giving place to a term whose counter-part in PPC would be equivalent to $\lambda_{[y]} \widehat{c}\, \widehat{y}\, \widehat{n}.y$ (*cf.* the reduction example in Sec. 2.1).

In the following sections $\mathsf{PPC}_{\mathtt{dB}}$ is shown to be equivalent to PPC in terms of expressive power and operational semantics. The main advantage of this new presentation is that it gets rid of α-conversion, since there is no possible collision between free and bound variables/matchables. However, there is one minor drawback with respect to the use of de Bruijn indices for the standard λ-calculus. As mentioned above, when working with de Bruijn indices in the standard λ-calculus, α-equivalence becomes syntactical equality.

Unfortunately, this is not the case when working with bidimensional indices. For instance, consider the terms $\lambda_2 \widehat{\mathtt{1}}_1\, \widehat{\mathtt{1}}_2.\mathtt{1}_1$ and $\lambda_2 \widehat{\mathtt{1}}_2\, \widehat{\mathtt{1}}_1.\mathtt{1}_2$. Both represent the function that decomposes an application and projects its first component. But they differ in the way the secondary indices are assigned. Moreover, one may be tempted to impose an order for the way the secondary indices are assigned within the pattern to avoid this situation (recall that the post-condition of the matching operation forces all bound symbols to appear in the pattern). Given the dynamic nature of patterns in the PPC framework, this enforcement would not solve the problem since patterns may reduce and such an order is not closed under reduction. For example, consider $\lambda_2(\lambda_2 \widehat{\mathtt{1}}_1\, \widehat{\mathtt{1}}_2.\mathtt{1}_2\, \mathtt{1}_1)\,(\widehat{\mathtt{1}}_1\, \widehat{\mathtt{1}}_2).\mathtt{1}_1 \rightarrow_{\mathtt{dB}} \lambda_2 \widehat{\mathtt{1}}_2\, \widehat{\mathtt{1}}_1.\mathtt{1}_1$.

Fortunately enough, this does not represent a problem from the implementation point of view, since the ambiguity is local to a binder and does not imply the need for "renaming" variables/matchables while reducing a term, *i.e.* no possible undesired capture can happen because of it. It is important to note though, that in the sequel, when refering to equality over terms of $\mathsf{PPC}_{\mathtt{dB}}$, it is not syntactical equality but equality modulo these assignments for secondary indices that we are using.

4 Translation

This section introduces translations between PPC and $\mathsf{PPC}_{\mathtt{dB}}$ (back and forth). The goal is to show that these interpretations are suitable to simulate one calculus into the other. Moreover, the proposed translations turn out to be the inverse of each other (modulo α-conversion) and, as we will see in Sec. 5, they allow to formalise a strong bisimulation between the two calculi.

We start with the translation from PPC to $\mathsf{PPC}_{\mathtt{dB}}$. It takes the term to be translated together with two lists of lists of symbols that dictate how the variables and matchables of the terms should be interpreted respectively. We use lists of lists since the first dimension indicates the distance to the binder, while the second identifies the symbol among the multiple bound ones.

Given the lists of lists X and Y, we denote by XY their concatenation. To improve readability, when it is clear from context, we also write θX with θ a list of symbols to denote $[\theta]X$ where $[\_]$ denotes the list constructor. We use set operations like union and intersection over lists to denote the union/intersection of its underlying sets.

Definition 4.1 Given a term $t \in \mathbb{T}_{\mathsf{PPC}}$ and lists of lists of symbols V and M such that $\mathsf{fv}(t) \subseteq \bigcup_{V' \in V} V'$ and $\mathsf{fm}(t) \subseteq \bigcup_{M' \in M} M'$, the *translation of* t *relative to* V *and* M, written $[\![t]\!]^M_V$, is inductively defined as follows:

$$\begin{array}{rcll}
[\![x]\!]^M_V & \triangleq & \mathtt{i}_\mathtt{j} & \text{where } i = \min\{i' \mid x \in V_{i'}\} \text{ and } j = \min\{j' \mid x = V_{ij'}\} \\
[\![\widehat{x}]\!]^M_V & \triangleq & \widehat{\mathtt{i}}_\mathtt{j} & \text{where } i = \min\{i' \mid x \in M_{i'}\} \text{ and } j = \min\{j' \mid x = M_{ij'}\} \\
[\![t\,u]\!]^M_V & \triangleq & [\![t]\!]^M_V\, [\![u]\!]^M_V & \\
[\![\lambda_\theta p.t]\!]^M_V & \triangleq & \lambda_{|\theta|} [\![p]\!]^{\theta M}_V.[\![t]\!]^M_{\theta V} &
\end{array}$$

Let $x_1, x_2, \ldots$ be an enumeration of $\mathbb{V}$. Then, the *translation* of t to $\mathsf{PPC}_{\mathtt{dB}}$, written simply $[\![t]\!]$, is defined as $[\![t]\!]^X_X$ where $X = [[x_1], \ldots, [x_n]]$ such that $\mathsf{fv}(t) \cup \mathsf{fm}(t) \subseteq \{x_1, \ldots, x_n\}$.

For example, consider the term $s_0 = (\lambda_{[x]} \widehat{y}\, \widehat{x}.x)\,(\widehat{y}\, s'_0)$ with $\mathsf{fv}(s'_0) = \{y\}$ and $\mathsf{fm}(s'_0) = \{y\}$. Then, $[\![s_0]\!]^{[[y]]}_{[[y]]} = (\lambda_1 [\![\widehat{y}]\!]^{[[x],[y]]}_{[[y]]}\, [\![\widehat{x}]\!]^{[[x],[y]]}_{[[y]]}.[\![x]\!]^{[[y]]}_{[[x],[y]]})\,([\![\widehat{y}]\!]^{[[y]]}_{[[y]]}\, [\![s'_0]\!]^{[[y]]}_{[[y]]}) = (\lambda_1 \widehat{\mathtt{2}}_1\, \widehat{\mathtt{1}}_1.\mathtt{1}_1)\,(\widehat{\mathtt{1}}_1\, [\![s'_0]\!]^{[[y]]}_{[[y]]})$. Note that the inicialisation of V and M with singleton elements implies that each free variable/matchable in the term will be assigned a distinct primary index (when interpreted at the same depth), following de Bruijn's original ideas: let $s_1 = (\lambda_{[x]} \widehat{y}\, \widehat{x}.x)\,(\widehat{y}\, z)$, then $[\![s_1]\!]^{[[y],[z]]}_{[[y],[z]]} = (\lambda_1 \widehat{\mathtt{2}_1}\, \widehat{\mathtt{1}_1}.\mathtt{1}_1)\,(\widehat{\mathtt{1}_1}\, \mathtt{2}_1)$.

Our main goal is to prove that $\mathsf{PPC_{dB}}$ simulates PPC via this embedding. For this purpose we need to state first some auxiliary lemmas that prove how the translation behaves with respect to the substitution and the matching operation. We start with a technical result concerning the increment functions for variable and matchable indices. Notation $\uparrow^n_k(t)$ stands for n consecutive applications of $\uparrow_k(\_)$ over t (similarly for $\Uparrow^n_k(t)$).

Lemma 4.2 *Let $t \in \mathbb{T}_{\mathsf{PPC}}$, $k \geq 0$, $i \geq 1$ and $n \geq k+i$ such that $X_l \cap (\mathsf{fv}(t) \cup \mathsf{fm}(t)) = \emptyset$ for all $l \in [k+1, k+i-1]$. Then,*

(i) $[\![t]\!]^M_{X_1 \ldots X_n} = \uparrow^{i-1}_k([\![t]\!]^M_{X_1 \ldots X_k X_{k+i} \ldots X_n})$.

(ii) $[\![t]\!]^{X_1 \ldots X_n}_V = \Uparrow^{i-1}_k([\![t]\!]^{X_1 \ldots X_k X_{k+i} \ldots X_n}_V)$.

The translation of a substitution σ requires an enumeration θ such that $\mathsf{dom}(\sigma) \subseteq \theta$ to be provided. It is then defined as $[\![\sigma, \theta]\!]^M_V \triangleq \{\mathtt{1_j} \setminus [\![\sigma x_j]\!]^M_V\}_{x_j \in \mathsf{dom}(\sigma)}$. Note how substitutions from PPC are mapped into substitutions at level 1 in the $\mathsf{PPC_{dB}}$ framework. This suffices since substitutions are only meant to be created in the context of a redex. When acting at arbitrary depth on a term, substitutions are shown to behave properly.

Lemma 4.3 *Let $s \in \mathbb{T}_{\mathsf{PPC}}$, σ be a substitution, θ be an enumeration such that $\mathsf{dom}(\sigma) \subseteq \theta$ and Y be a list of $i-1$ lists of symbols such that $(\bigcup_{Y' \in Y} Y') \cap \theta = \emptyset$ and $(\bigcup_{Y' \in Y} Y') \cap (\bigcup_{x_j \in \theta} \mathsf{fv}(\sigma x_j)) = \emptyset$. Then, $[\![\sigma s]\!]^M_{YX} = \downarrow_{i-1}(\{\mathtt{i_j} \setminus \uparrow_{i-1}([\![\sigma x_j]\!]^M_{YX})\}_{x_j \in \mathsf{dom}(\sigma)} [\![s]\!]^M_{Y\theta X})$.*

In the case of a match, its translation is given by $[\![\{\!\{p \setminus^{\theta} u\}\!\}]\!]^M_V \triangleq \{\!\{[\![p]\!]^{\theta M}_V \setminus^{|\theta|} [\![u]\!]^M_V\}\!\}$. Note how θ is pushed into the matchable symbol list of the pattern, in accordance with the translation of an abstraction. This is crucial for the following result of preservation of the matching output.

Lemma 4.4 *Let $p, u \in \mathbb{T}_{\mathsf{PPC}}$.*

(i) *If $\{\!\{p \setminus^{\theta} u\}\!\} = \sigma$, then $[\![\{\!\{p \setminus^{\theta} u\}\!\}]\!]^M_V = [\![\sigma, \theta]\!]^M_V$.*

(ii) *If $\{\!\{p \setminus^{\theta} u\}\!\} = \mathtt{fail}$, then $[\![\{\!\{p \setminus^{\theta} u\}\!\}]\!]^M_V = \mathtt{fail}$.*

(iii) *If $\{\!\{p \setminus^{\theta} u\}\!\} = \mathtt{wait}$, then $[\![\{\!\{p \setminus^{\theta} u\}\!\}]\!]^M_V = \mathtt{wait}$.*

These previous results will allow to prove the simulation of PPC into $\mathsf{PPC_{dB}}$ via the translation $[\![\_]\!]$. We postpone this result to Sec. 5 (*cf.* Thm. 5.1).

Now we focus on the converse side of the embedding, *i.e.* translation of $\mathsf{PPC_{dB}}$ terms into PPC terms. As before, this mapping requires two lists of lists of symbols from which names of the free indices of the term will be selected: one for variable indices and the other for matchable indices.

Definition 4.5 Given a term $t \in \mathbb{T}_{\mathsf{PPC_{dB}}}$ and lists of lists of distinct symbols V and M such that V_{ij} is defined for every $\mathtt{i_j} \in \mathsf{fv}(t)$ and M_{ij} is defined for every $\mathtt{i_j} \in \mathsf{fm}(t)$, the *translation of t relative to V and M*, written $(\!|t|\!)^M_V$, is inductively defined as follows:

$$
\begin{array}{rcll}
(\!|\mathtt{i_j}|\!)^M_V & \triangleq & V_{ij} & \\
(\!|\widehat{\mathtt{i_j}}|\!)^M_V & \triangleq & \widehat{M_{ij}} & \\
(\!|t\,u|\!)^M_V & \triangleq & (\!|t|\!)^M_V\ (\!|u|\!)^M_V & \\
(\!|\lambda_n p.t|\!)^M_V & \triangleq & \lambda_\theta (\!|p|\!)^{\theta M}_V.(\!|t|\!)^M_{\theta V} & \theta = [x_1, \ldots, x_n] \text{ fresh symbols}
\end{array}
$$

Let $x_1, x_2, \ldots$ be the same enumeration of $\mathbb{V}$ as in Def. 4.1. Then, the *translation* of t to PPC, written simply $(\!|t|\!)$, is defined as $(\!|t|\!)^X_X$ where $X = [[x_1], \ldots, [x_n]]$ such that $\mathsf{fv}(t) \cup \mathsf{fm}(t) \subseteq \{\mathtt{1}_1, \ldots, \mathtt{n}_1\}$. Note that well-formedness of terms guarantees that X satisfies the conditions above.

To illustrate the translation, consider the term $t_1 = (\lambda_1 \widehat{\mathtt{2}_1}\, \widehat{\mathtt{1}_1}.\mathtt{1}_1)\,(\widehat{\mathtt{1}_1}\, \mathtt{2}_1)$ where $\mathsf{fv}(t_1) = \{\mathtt{2}_1\}$ and $\mathsf{fm}(t_1) = \{\mathtt{1}_1\}$. Then, $(\!|t_1|\!)^{[[y],[z]]}_{[[y],[z]]} = (\lambda_{[x]} (\!|\widehat{\mathtt{2}_1}\, \widehat{\mathtt{1}_1}|\!)^{[[x],[y],[z]]}_{[[y],[z]]}.(\!|\mathtt{1}_1|\!)^{[[y],[z]]}_{[[x],[y],[z]]})\,(\!|\widehat{\mathtt{1}_1}\, \mathtt{2}_1|\!)^{[[y],[z]]}_{[[y],[z]]} = (\lambda_{[x]} \widehat{y}\, \widehat{x}.x)\,(\widehat{y}\, z)$. Note that $t_1 = [\![s_1]\!]$ from the example after Def. 4.1 and, with a proper initialisation of the lists V and M, we get $(\!|t_1|\!) = s_1$.

Once again, we start with some technical lemmas for substitutions and the matching operations with respect to the embedding $\llparenthesis \_ \rrparenthesis$. In this case, the increment functions for variable and matchable indices behave as follows:

Lemma 4.6 *Let $t \in \mathbb{T}_{\mathsf{PPC}_{\mathsf{dB}}}$, $k \geq 0$, $i \geq 1$ and $n \geq k+i$ such that $\mathsf{fv}(t) \cup \mathsf{fm}(t) \subseteq \{\mathtt{i'}_{\mathtt{j}} \mid i' \leq n-(i-1), j \leq |X_{i'}|\}$. Then,*

(i) $\llparenthesis \uparrow_k^{i-1}(t) \rrparenthesis^M_{X_1 \ldots X_n} =_\alpha \llparenthesis t \rrparenthesis^M_{X_1 \ldots X_k X_{k+i} \ldots X_n}$.

(ii) $\llparenthesis \Uparrow_k^{i-1}(t) \rrparenthesis^{X_1 \ldots X_n}_V =_\alpha \llparenthesis t \rrparenthesis^{X_1 \ldots X_k X_{k+i} \ldots X_n}_V$.

As for the converse, the translation is only defined for substitution at level 1 and requires to be provided a list of symbols θ such that $|\theta| \geq \max\{j \mid \mathtt{1}_{\mathtt{j}} \in \mathsf{dom}(\sigma)\}$. Then, $\llparenthesis \sigma, \theta \rrparenthesis^M_V \triangleq \{\theta_j \backslash \llparenthesis \sigma \mathtt{1}_{\mathtt{j}} \rrparenthesis^M_V\}_{\mathtt{1}_{\mathtt{j}} \in \mathsf{dom}(\sigma)}$. The application of a substitution at an arbitrary level i is shown to translate properly.

Lemma 4.7 *Let $s \in \mathbb{T}_{\mathsf{PPC}_{\mathsf{dB}}}$, σ be a substitution at level i, θ be a list of fresh symbols such that $|\theta| \geq \max\{j \mid \mathtt{i}_{\mathtt{j}} \in \mathsf{dom}(\sigma)\}$ and Y be a list of $i-1$ lists of symbols. Then, $\llparenthesis \downarrow_{i-1}(\{\mathtt{i}_{\mathtt{j}} \backslash \uparrow_{i-1}(\sigma \mathtt{i}_{\mathtt{j}})\}_{\mathtt{i}_{\mathtt{j}} \in \mathsf{dom}(\sigma)} s) \rrparenthesis^M_{YX} =_\alpha \{\theta_j \backslash \llparenthesis \sigma \mathtt{i}_{\mathtt{j}} \rrparenthesis^M_{YX}\}_{\mathtt{i}_{\mathtt{j}} \in \mathsf{dom}(\sigma)} \llparenthesis s \rrparenthesis^M_{Y\theta X}$.*

Similarly to the substitution case, the translation of a match $\{\!\{p \backslash^n u\}\!\}$ requires to be supplied with a list of n fresh symbols θ. Then, it is defined as $\llparenthesis \{\!\{p \backslash^n u\}\!\}, \theta \rrparenthesis^M_V \triangleq \{\!\{\llparenthesis p \rrparenthesis^{\theta M}_V \backslash^\theta \llparenthesis u \rrparenthesis^M_V\}\!\}$. The newly provided list of symbols is used both as the parameter of the resulting match and to properly translate the pattern, obtaining the following expected result.

Lemma 4.8 *Let $p, u \in \mathbb{T}_{\mathsf{PPC}_{\mathsf{dB}}}$.*

(i) *If $\{\!\{p \backslash^n u\}\!\} = \sigma$, then $\llparenthesis \{\!\{p \backslash^n u\}\!\}, \theta \rrparenthesis^M_V = \llparenthesis \sigma, \theta \rrparenthesis^M_V$.*

(ii) *If $\{\!\{p \backslash^n u\}\!\} = \mathtt{fail}$, then $\llparenthesis \{\!\{p \backslash^n u\}\!\}, \theta \rrparenthesis^M_V = \mathtt{fail}$.*

(iii) *If $\{\!\{p \backslash^n u\}\!\} = \mathtt{wait}$, then $\llparenthesis \{\!\{p \backslash^n u\}\!\}, \theta \rrparenthesis^M_V = \mathtt{wait}$.*

Now we are in conditions to prove the simulation of $\mathsf{PPC}_{\mathsf{dB}}$ into PPC via the translation provided in Def. 4.5.

Before proceeding to the next section, one final result concerns the translations. It turns out that each translation is the inverse of the other, as shown in Thm. 4.9. In case of PPC terms we should work modulo α-conversion, while for $\mathsf{PPC}_{\mathsf{dB}}$ terms we may use equality (modulo secondary indices permutations, *cf.* last paragraph in Sec. 3). This constitutes the main result of this section and is the key to extend our individual simulation results (*cf.* Thm. 5.1 and 5.2 resp.) into a strong bisimulation between the two calculi, as shown in Sec. 5.

Theorem 4.9 (Invertibility) *Let $t \in \mathbb{T}_{\mathsf{PPC}_{\mathsf{dB}}}$ and $s \in \mathbb{T}_{\mathsf{PPC}}$. Then,* (i) *$[\![\llparenthesis t \rrparenthesis]\!] = t$; and* (ii) *$\llparenthesis [\![s]\!] \rrparenthesis =_\alpha s$.*

Proof By straightforward induction on t and s respectively. Details in [19]. □

5 Strong bisimulation

In this section we prove the simulation of one calculus by the other via the proper translation and, most importantly, the strong bisimulation that follows after the invertibility result (*cf.* Thm. 4.9). This strong bisimulation result will allow to port many important properties already known for PPC into $\mathsf{PPC}_{\mathsf{dB}}$, as we will discuss later.

We start by simulating PPC by $\mathsf{PPC}_{\mathsf{dB}}$. The key step here is the preservation of the matching operation shown for $[\![\_]\!]$ in Lem. 4.4. It guarantees that every redex in PPC turns into a redex in $\mathsf{PPC}_{\mathsf{dB}}$ too. Then, the appropiate definition of the operational semantics given for $\mathsf{PPC}_{\mathsf{dB}}$ in Sec. 3 allows us to conclude.

Theorem 5.1 *Let $t \in \mathbb{T}_{\mathsf{PPC}}$. If $t \to_{\mathsf{PPC}} t'$, then $[\![t]\!] \to_{\mathsf{dB}} [\![t']\!]$.*

Proof By induction on $t \to_{\mathsf{PPC}} t'$ using Lem. 4.2, 4.3 and 4.4. Details in [19]. □

Regarding the converse simulation, *i.e.* $\mathsf{PPC}_{\mathsf{dB}}$ into PPC, we resort here to the fact that the embedding $\llparenthesis \_ \rrparenthesis$ also preserves the matching operation (*cf.* Lem. 4.8). Then, every redex in $\mathsf{PPC}_{\mathsf{dB}}$ is translated into a redex in PPC as well.

Theorem 5.2 *Let $t \in \mathbb{T}_{\mathsf{PPC}_{\mathsf{dB}}}$. If $t \to_{\mathsf{dB}} t'$, then $\llparenthesis t \rrparenthesis \to_{\mathsf{PPC}} \llparenthesis t' \rrparenthesis$.*

Proof By induction on $t \to_{\mathsf{dB}} t'$ using Lem. 4.6, 4.7 and 4.8. Details in [19]. $\square$

As already commented, these previous results may be combined to obtain a strong bisimulation between the two calculi. The invertibility result allows to define a relation between terms in PPC and $\mathsf{PPC_{dB}}$. Given $t \in \mathbb{T}_{\mathsf{PPC_{dB}}}$ and $s \in \mathbb{T}_{\mathsf{PPC}}$, let us write $t \Mapsto s$ whenever $(\!| t |\!) =_\alpha s$ and, therefore, $[\![s]\!] = t$ by Thm. 4.9. Then, the strong bisimulation result states that whenever $t \Mapsto s$ and $t \to_{\mathsf{dB}} t'$, there exists a term s' such that $t' \Mapsto s'$ and $s \to_{\mathsf{PPC}} s'$, and the other way around. Graphically:

$$\begin{tikzcd}
t \arrow[r, phantom, "\Mapsto"] \arrow[d, "\mathsf{dB}"'] & s \arrow[d, dashed, "\mathsf{PPC}"'] & & t \arrow[r, phantom, "\Mapsto"] \arrow[d, dashed, "\mathsf{dB}"'] & s \arrow[d, "\mathsf{PPC}"'] \\
t' \arrow[r, phantom, "\Mapsto"] & s' & \text{and} & t' \arrow[r, phantom, "\Mapsto"] & s'
\end{tikzcd}$$

Theorem 5.3 (Strong bisimulation) *The relation $\Mapsto$ is a strong bisimulation with respect to the reduction relations $\to_{\mathsf{PPC}}$ and $\to_{\mathsf{dB}}$ respectively.*

Proof The proof follows immediately from Thm. 5.1 and 5.2 above, and the invertibility result (Thm. 4.9) given in Sec. 4. $\square$

The importance of this result resides in the fact that it guarantees that PPC and $\mathsf{PPC_{dB}}$ have exactly the same operational semantics. An immediate consequence of this is the confluence of $\to_{\mathsf{dB}}$.

Theorem 5.4 (Confluence) *The reduction relation $\to_{\mathsf{dB}}$ is confluent (CR).*

Proof The result follows directly from Thm. 2.1 and the strong bisimulation:

$$\begin{tikzcd}
 & & t \arrow[lld, two heads, "\mathsf{dB}"'] \arrow[rrd, two heads, "\mathsf{dB}"] \arrow[d, Rightarrow, "\Mapsto" description] & & \\
 & & s \arrow[ld, dashed, two heads, "\mathsf{PPC}"'] \arrow[rd, dashed, two heads, "\mathsf{PPC}"] & & \\
t_0 \arrow[r, phantom, "\Mapsto"] \arrow[rrdd, dashed, two heads, "\mathsf{dB}"'] & s_0 \arrow[rd, dashed, two heads, "\mathsf{PPC}"] & \text{Thm. 2.1} & s_1 \arrow[ld, dashed, two heads, "\mathsf{PPC}"'] & t_1 \arrow[l, phantom, "\Mapsfrom"] \arrow[lldd, dashed, two heads, "\mathsf{dB}"] \\
 & & s' & & \\
 & & t' \arrow[u, Rightarrow] & &
\end{tikzcd}$$

$\square$

Another result of relevance to our line of research, is the existence of normalising reduction strategies for PPC, as it is shown in [4]. This result is particularly challenging since reduction in PPC is shown to be *non-sequential* due to the nature of its matching operation. This implies that the notion of *needed redexes* (a key concept for defining normalising strategies) must be generalised to *necessary sets* [26] of redexes. Moreover, the notion of *gripping* [21] is also captured by PPC, representing a further obstacle in the definition of such a normalising strategy. All in all, in [4] the authors introduce a reduction strategy $\mathcal{S}$ that is shown to be normalising, overcoming all the aforementioned issues. Thanks to the strong bisimulation result presented above, this strategy $\mathcal{S}$ can also be guaranteed to normalise for $\mathsf{PPC_{dB}}$.

6 Conclusion

In this paper we introduced a novel presentation of the *Pure Pattern Calculus* (PPC) [15] in de Bruijn's style. This required extending de Bruijn ideas for a setting where each binder may capture more than one variable at once. To this purpose we defined *bidimensional indices* of the form $\mathtt{i_j}$ where $\mathtt{i}$ is dubbed the *primary index* and $\mathtt{j}$ the *secondary index*, so that the primary index determines the binding abstraction and the secondary index identifies the variable among those (possibly many) bound ones. Moreover, given the nature of PPC semantics, our extension actually deals with two kinds of bidimensional indices, namely *variable indices* and *matchable indices*. This newly introduced calculus is simply called *Pure Pattern Calculus with de Bruijn indices* ($\mathsf{PPC_{dB}}$).

Our main result consists of showing that the relation between PPC and $\mathsf{PPC_{dB}}$ is a strong bisimulation with respect to their respective redution relations, *i.e.* they have exactly the same operational semantics. For that reason, proper translations between the two calculi were defined, $[\![\_]\!] : \mathbb{T}_{\mathsf{PPC}} \to \mathbb{T}_{\mathsf{PPC_{dB}}}$ and $(\!|\_|\!) : \mathbb{T}_{\mathsf{PPC_{dB}}} \to \mathbb{T}_{\mathsf{PPC}}$, in such a way that $[\![\_]\!]$ is the inverse of $(\!|\_|\!)$ and vice-versa (modulo α-conversion). Most notably, these embeddings are shown to preserve the matching operation of their respective domain calculus.

The strong bisimulation result allows to port into $\mathsf{PPC}_{\mathsf{dB}}$ many already known results for PPC. Of particular interest for our line of research are the confluence and the existence of normalising reduction strategies for PPC [4], a rather complex result that requires dealing with notions of *gripping* [21] and *necessary sets* [26] of redexes. The result introduced on this paper may allow for a direct implementation of such strategies without the inconveniences of working modulo α-conversion.

As commented before, the ultimate goal of our research is the implementation of a prototype for a typed functional programming language capturing *path polymorphism*. This development is based on the *Calculus of Applicative Patterns* (CAP) [1], for which a static type system has already been introduced, guaranteeing well-behaved operational semantics, together with its corresponding efficient type-checking algorithm. CAP is essentially the static fragment of PPC, where the abstraction is generalised into an alternative (*i.e.* abstracting multiple branches at once). These two calculi are shown to be equivalent. Thus, future lines of work following the results presented in this paper involve porting CAP to the bidimensional indices setting and formalising the ideas of [4] into such framework. This will lead to a first functional version of the sought-after prototype.

References

[1] Ayala-Rincón, M., E. Bonelli, J. Edi and A. Viso, *Typed path polymorphism*, Theor. Comput. Sci. **781** (2019), pp. 111–130.
URL https://doi.org/10.1016/j.tcs.2019.02.018

[2] Barendregt, H. P., "The lambda calculus - its syntax and semantics," Studies in logic and the foundations of mathematics **103**, North-Holland, 1985.

[3] Bonelli, E., D. Kesner, C. Lombardi and A. Ríos, *Normalisation for dynamic pattern calculi*, in: A. Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, LIPIcs **15** (2012), pp. 117–132.
URL https://doi.org/10.4230/LIPIcs.RTA.2012.117

[4] Bonelli, E., D. Kesner, C. Lombardi and A. Ríos, *On abstract normalisation beyond neededness*, Theor. Comput. Sci. **672** (2017), pp. 36–63.
URL https://doi.org/10.1016/j.tcs.2017.01.025

[5] Bonelli, E., D. Kesner and A. Ríos, *de Bruijn indices for metaterms*, J. Log. Comput. **15** (2005), pp. 855–899.
URL https://doi.org/10.1093/logcom/exi051

[6] Cerrito, S. and D. Kesner, *Pattern matching as cut elimination*, Theor. Comput. Sci. **323** (2004), pp. 71–127.
URL https://doi.org/10.1016/j.tcs.2004.03.032

[7] Cirstea, H. and C. Kirchner, *ρ-calculus. Its Syntax and Basic Properties*, in: *CCL*, 1998, pp. 66–85.

[8] de Bruijn, N. G., *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem*, Indagationes Mathematicae **75** (1972), pp. 381–392.

[9] de Bruijn, N. G., "A namefree lambda calculus with facilities for internal definition of expressions and segments," EUT report. WSK, Dept. of Mathematics and Computing Science, Technische Hogeschool Eindhoven, 1978.

[10] Edi, J., A. Viso and E. Bonelli, *Efficient type checking for path polymorphism*, in: T. Uustalu, editor, *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, LIPIcs **69** (2015), pp. 6:1–6:23.
URL https://doi.org/10.4230/LIPIcs.TYPES.2015.6

[11] Glauert, J. R. W., D. Kesner and Z. Khasidashvili, *Expression reduction systems and extensions: An overview*, in: A. Middeldorp, V. van Oostrom, F. van Raamsdonk and R. C. de Vrijer, editors, *Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science **3838** (2005), pp. 496–553.
URL https://doi.org/10.1007/11601548_22

[12] Jay, B., *The pattern calculus*, ACM Trans. Program. Lang. Syst. **26** (2004), pp. 911–937.
URL https://doi.org/10.1145/1034774.1034775

[13] Jay, B., "Pattern Calculus - Computing with Functions and Structures," Springer, 2009.
URL https://doi.org/10.1007/978-3-540-89185-7

[14] Jay, B. and D. Kesner, *Pure pattern calculus*, in: P. Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, Lecture Notes in Computer Science **3924** (2006), pp. 100–114.
URL https://doi.org/10.1007/11693024_8

[15] Jay, B. and D. Kesner, *First-class patterns*, J. Funct. Program. **19** (2009), pp. 191–225.
URL https://doi.org/10.1017/S0956796808007144

[16] Kahl, W., *Basic pattern matching calculi: Syntax, reduction, confluence, and normalisation* (2003).

[17] Kamareddine, F. and A. Ríos, *A lambda-calculus à la de Bruijn with explicit substitutions*, in: M. V. Hermenegildo and S. D. Swierstra, editors, *Programming Languages: Implementations, Logics and Programs, 7th International Symposium, PLILP'95, Utrecht, The Netherlands, September 20-22, 1995, Proceedings*, Lecture Notes in Computer Science **982** (1995), pp. 45–62.
URL https://doi.org/10.1007/BFb0026813

[18] Klop, J. W., V. van Oostrom and R. C. de Vrijer, *Lambda calculus with patterns*, Theor. Comput. Sci. **398** (2008), pp. 16–31.

[19] Martín, A., A. Ríos and A. Viso, *Pure pattern calculus* à la *de Bruijn*, Extended report (2020), https://arxiv.org/abs/2006.07674.

[20] Mayr, R. and T. Nipkow, *Higher-order rewrite systems and their confluence*, Theor. Comput. Sci. **192** (1998), pp. 3–29.
URL https://doi.org/10.1016/S0304-3975(97)00143-6

[21] Melliès, P.-A., "Description Abstraite des Systèmes de Réécriture," Ph.D. thesis, Université Paris VII (1996).

[22] Nipkow, T., *Higher-order critical pairs*, in: *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991* (1991), pp. 342–349.
URL https://doi.org/10.1109/LICS.1991.151658

[23] van Oostrom, V., *Lambda calculus with patterns*, Technical Report IR-228, Vrije Universiteit, Amsterdam (1990).

[24] van Oostrom, V. and F. van Raamsdonk, *The dynamic pattern calculus as a higher-order pattern rewriting system*, in: *7th International Workshop on Higher-Order Rewriting, HOR 2014, Held as Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 9-24, 2014. Proceedings*, 2014.

[25] van Raamsdonk, F., "Confluence and Normalization for Higher-Order Rewriting," Ph.D. thesis, Amsterdam University (1996).

[26] van Raamsdonk, F., *Outermost-fair rewriting*, in: P. de Groote, editor, *Typed Lambda Calculi and Applications, Third International Conference on Typed Lambda Calculi and Applications, TLCA '97, Nancy, France, April 2-4, 1997, Proceedings*, Lecture Notes in Computer Science **1210** (1997), pp. 284–299.
URL https://doi.org/10.1007/3-540-62688-3_42

[27] Viso, A., "Un estudio semántico sobre extensiones avanzadas del λ-cálculo: patrones y operadores de control," Ph.D. thesis, Universidad de Buenos Aires (2020).

[28] Viso, A., E. Bonelli and M. Ayala-Rincón, *Type soundness for path polymorphism*, in: M. R. F. Benevides and R. Thiemann, editors, *Proceedings of the Tenth Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2015, Natal, Brazil, August 31 - September 1, 2015*, Electronic Notes in Theoretical Computer Science **323** (2015), pp. 235–251.
URL https://doi.org/10.1016/j.entcs.2016.06.015

EVL: a typed higher-order functional language for events

Sandra Alves[1]

DCC-FCUP & CRACS
University of Porto, Porto, Portugal

Maribel Fernández[2]

Dept. of Informatics
King's College London, London WC2B 4BG, U.K.

Miguel Ramos[3]

DCC-FCUP & CRACS
University of Porto, Porto, Portugal

Abstract

We define EVL, a minimal higher-order functional language for dealing with generic events. The notion of generic event extends the well-known notion of event traditionally used in a variety of areas, such as database management, concurrency, reactive systems and cybersecurity. Generic events were introduced in the context of a metamodel to deal with obligations in access control systems. Event specifications are represented as records and we use polymorphic record types to type events in our language. We show how the higher-order capabilities of EVL can be used in the context of Complex Event Processing (CEP), to define higher-order parameterised functions that deal with the usual CEP techniques.

Keywords: events, access control, obligations, record types.

1 Introduction

In today's complex systems, where information is constantly being generated, there is a pressing need for efficiently processing data, and in particular data related with actions taking place in a system. Occurrences of actions, or happenings, are known as *events*, and technology for processing event streams, referred to as Complex Event Processing (CEP), has been around for decades [6,23,15]. Most of the CEP systems focus on real-life application therefore investing in issues such as scalability, fault tolerance, distribution, amongst others, but often lacking a formal semantics, making them difficult to understand, extend or generalize.

In the context of security, and in particular when modeling access control, it is often the case that granting or denying access to certain resources depends on the occurrence of a particular event [7,21,8]. This is even more crucial in access control systems dealing with obligations, where the status of a particular obligations is usually defined in terms of event occurrences in the system, and several models that deal with obligations have to deal in some way with the notion of event. The Category-Based metatamodel for Access Control and Obligations (CBACO [2]), defines an axiomatisation of the notion of obligation based on generic relations to type events and extract event intervals. A key notion in that model is to distinguish between event schemes,

[1] Email: sandra@fc.up.pt

[2] Email: maribel.fernandez@kcl.ac.uk

[3] Email: jmiguelsramos@gmail.com

which provide a general description of the kind of events that can occur in a particular system, and specific events, which describe actual events that have occurred. Note that events can take various forms, depending on the system that is being considered (for example, messages exchanged over a network, actions performed by users of the system, occurrences of physical phenomena such as a disk error or a fire alarm, etc). To deal with event classification in a uniform way, Alves et. al. [1] defined a general term-based language for events. In this language, events were presented as typed-terms, built from a user-defined signature, that is, a particular set of typed function symbols that are specific to the system modelled. With this approach it is possible to define general functions to implement event typing and to compute event intervals, without needing to know the exact type of events.

In that context, a compound event [1] links a set of events that can occur separately in the history, but should be identified as a single event occurrence. For simplicity, in [1] compound events were assumed to appear as a single event in history, leaving a more detailed and realistic treatment of compound events for future work. This notion of compound or composite event is also a key feature in CEP systems, which put great emphasis on the ability to detect complex patterns of incoming streams of events and establish sequencing and ordering relations.

Types were used in [1] not only to ensure that terms representing events respect the type signature specific to the system under study, but also to formally define the notion of event instantiation, associating specific events to generic events through an implicit notion of subtyping, inspired by Ohori's system of polymorphic record types [24]. Because of the implicit subtyping rule for typing records, the system defined in [1] allowed for type-checking of event-specification, but not for dealing with most general types for event specifications.

In this paper we take a step further and define `EVL`, a higher-order polymorphic typed language, designed to facilitate the specification and processing of events. Our language is both a restriction and an extension of Ohori's polymorphic record calculus [24], and although our type system is very much based on that system, it is not meant to be a general language, but rather a language purposely designed for dealing with events. Languages traditionally used in event processing systems are usually derived from relational languages, in particular, relational algebra and SQL, extended with additional ad-hoc operators to better support information flow or imperative programming. This paper explores the potential of the functional paradigm in this context, both at the level of the type-system, as well as exploring the higher-order capabilities of the language. The main contributions of this paper are:

- The design of `EVL`: a higher-order typed polymorphic record calculus for event processing. Our goal is to have a minimal language, but which is expressive enough to specify and manipulate events.
- A sound and complete type inference algorithm for `EVL`, defined as both an extension and a restriction of the ML-style record calculus in [24].
- A comprehensive study of the `EVL` higher-order/functional capabilities and its application in the context of CEP.

Overview

In Section 2 we define the `EVL` language and its set of types. In Section 3 we define a type system for `EVL` and in Section 4 we present a type inference algorithm, which is proved to be sound and complete. In Section 5 we discuss CEP and explore `EVL`'s capabilities in this context. We discuss related work in Section 6 and we finally conclude and discuss further work in Section 7.

2 The `EVL` typed language

In this section we introduce `EVL`, a minimalistic typed language to specify events. `EVL` is an extension of the λ-calculus that includes records, a flexible data structure that is used here to deal with event specifications as defined in [1]. More precisely, an event specification is a labelled structure (or record) of the form $\{l_1 = v_1, \ldots, l_n = v_n\}$, representing a set of labels $l_1, \ldots, l_n$ with associated values $v_1, \ldots, v_n$. We assume some familiarity with the λ-calculus (see [5] for a detailed reference).

2.1 Terms

We start by formally defining the set of `EVL` terms. In the following, let $x, y, z, \ldots$ range over a countable set of variables and $l, l_1, \ldots$ range over a countable set $\mathcal{L}$ of labels.

Definition 2.1 The set of EVL terms is given by the following grammar:

$$
\begin{aligned}
M ::= {} & x \mid MM \mid \lambda x.M \mid \text{if } M \text{ then } M \text{ else } M \\
& \text{let } x = M \text{ in } M \mid \text{letEv } x = M \text{ in } M \\
& \{l = M, \dots, l = M\} \mid M.l \mid \text{modify}(M, l, M)
\end{aligned}
$$

Notation: We will use the notation *let* $x\ x_1 \dots x_n = M$ and *letEv* $x\ x_1 \dots x_n = M$ for *let* $x = \lambda x_1 \dots \lambda x_n.M$ and *letEv* $x = \lambda x_1 \dots \lambda x_n.M$, respectively. As an abuse of notation, in examples, we will use more meaningful names for functions, labels and events. Furthermore, event names will always start with a capital letter, to help distinguish them from functions.

We choose not to add other potentially useful constructors to the language, for instance pairs and projections, since we are aiming at a minimal language. Nevertheless, we can easily encode pairs (M_1, M_2) and projections $\pi_1 M$ and $\pi_2 M$ in our language by means of records of the form $\{\text{fst} = M_1, \text{snd} = M_2\}$ and M.fst, M.snd, respectively. This can trivially be extended to tuples in general, and we will often use this notation when writing examples.

Example 2.2 In this simple example, *FireDanger* reports the fire danger level of a particular location.

```
letEv FireDanger = λlλd.{location = l, fire_danger = d} in
FireDanger "Porto" "low"
```

To make our examples more readable, we will also use the following terms, abbreviating list construction:

```
nil = {empty = true}

cons x list = {empty = false, head = x, tail = list}.
```

Note that, much like what happens with tuples, the type of a particular list in this notation will be closely related to the size of the list in question. A more realistic approach is to add lists and list-types as primitive notions in the language, but, as we mentioned before, we are focusing on a minimal language. Furthermore, we will often use constants (numbers, booleans, strings, etc) and operators (arithmetic, boolean, etc) in our examples. However, following the minimalistic approach, we do not add constants/operators to the grammar and instead use free variables to represent them. Again, in a more general approach we could extend the grammar with other data structures and operators, for numbers, booleans, lists, etc.

Example 2.3 Consider the following example illustrating the definition of a generic event *FireDanger* and of a function *check* that determines if there is the danger of a fire erupting in a particular location, using the weather information associated with that location. Function *check* creates an appropriate instance of *FireDanger* to report the appropriate fire danger level.

```
letEv FireDanger l d = {location = l, fire_danger = d} in
let check x = if (x.temperature > 29 and x.wind > 32
                  and x.humidity < 20 and x.precipitation < 50)
              then FireDanger x.location "high"
              else FireDanger x.location "low" in
    check {temperature = 10, wind = 20, humidity = 30,
           precipitation = 10, location = "Porto"}
```

2.2 Types

We now define the set of types, and a typing system for the EVL language. We use record types to type labelled structures following Ohori's ML-system with polymorphic record types [24]. This extends the standard type system for parametric polymorphism by Damas and Milner [13]. In Ohori's system, polymorphic types are defined by type schemes of the form $\forall\alpha :: \kappa.\sigma$, where the type variable α is restricted to a set of types κ called a kind. We assume a finite set $\mathbb{B}$ of constant types and a countable set $\mathbb{V}$ of type variables, and we will use $b, b_1, \dots, \alpha, \alpha_1, \dots$ and $\kappa, \kappa_1, \dots$ to denote constant types, type variables and kinds, respectively. The set $\mathbb{B}$ of constant types will always contain the type *bool*.

Definition 2.4 The set of types σ and kinds κ are specified by the following grammar.

$$
\begin{aligned}
\sigma &::= \tau \mid \forall\alpha :: \kappa.\sigma \\
\tau &::= \alpha \mid b \mid \tau \to \tau \mid \{l : \tau, \ldots, l : \tau\} \\
\rho &::= \alpha \mid b \mid \tau \to \rho \\
\gamma &::= \tau \to \{l : \rho, \ldots, l : \rho\} \mid \{l : \rho, \ldots, l : \rho\} \\
\kappa &::= \mathcal{U} \mid \{\{l : \tau, \ldots, l : \tau\}\}
\end{aligned}
$$

Following Damas and Milner's type system, we divide the set of types into *monotypes* (ranged over by τ) and *polytypes* (of the form $\forall\alpha :: \kappa.\sigma$). More precisely, σ represents all types and τ represents all monotypes. We denote by ρ (included in τ) the type of event fields, and by γ (also included in τ) the type of event definitions. This distinction is necessary to adequately type event definitions and its purpose will become clear in the definition of the typing system. We use $\mathcal{U}$ for the kind representing every possible type, and the kind $\{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\}$ represents record types that contain, at least, the fields $l_1, \ldots, l_n$, with types $\tau_1, \ldots, \tau_n$, respectively.

We do not allow nested events, and to that end we clearly separate types for event definitions, denoted by γ, and which are a subset of the general types denoted by τ. However, we do allow for nested records of general type. The following is an example of a term that is typed with a nested record type:

```
{empty = false , head = 1, tail = {empty = false , head = 2, tail = {empty = true} } }.
```

Let F range over functions from a finite set of labels to types. We write $\{F\}$ and $\{\{F\}\}$ to denote the record type identified by F and the record kind identified by F, respectively. For two functions F_1 and F_2 we write $F_1 \pm F_2$ for the function F such that $dom(F) = dom(F_1) \cup dom(F_2)$ and such that for $l \in dom(F)$, $F(l) = F_1(l)$ if $l \in dom(F_1)$; otherwise $F(l) = F_2(l)$.

Notation: Following the notation for pairs introduced above, we write $(\sigma_1 \times \sigma_2)$ for the product type corresponding to $\{\text{fst} : \sigma_1, \text{snd} : \sigma_2\}$.

A typing environment Γ is a set of statements $x : \sigma$ where all subjects x are distinct. We write $dom(\Gamma)$ to denote the domain of a typing environment $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$, which is the set $\{x_1, \ldots, x_n\}$. The type of a variable $x_i \in dom(\Gamma)$ is $\Gamma(x_i) = \sigma_i$, and we write Γ_x to denote $\Gamma \setminus \{x : \Gamma(x)\}$. A kinding environment K is a set of statements $\alpha :: \kappa$. Similarly, the domain of a kinding environment $K = \{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\}$, denoted $dom(K)$, is the set $\{\alpha_1, \ldots, \alpha_n\}$ and the kind of a type $\alpha_i \in dom(K)$ is $K(\alpha_i) = \kappa_i$. A type variable α occurring in a type/kind is bound, if it occurs under the scope of a $\forall$-quantifier on α, otherwise it is free. We denote by $FTV(\sigma)$ ($FTV(\kappa)$) the set of free variables of σ (respectively, κ). We say that a type σ and a kind κ are well-formed under a kinding environment K if $FTV(\sigma) \subseteq dom(K)$ and $FTV(\kappa) \subseteq dom(K)$, respectively. A typing environment Γ is well-formed under a kinding environment K, if $\forall x \in dom(\Gamma)$, $\Gamma(x)$ is well-formed under K. A kinding environment K is well-formed, if $\forall\alpha \in dom(K), FTV(K(\alpha)) \subseteq dom(K)$. This reflects the fact that every free type variable in an expression has to be restricted by a kind in the kinding environment. Therefore, every type variable is either restricted by the kind in the type scheme or by a kind in the kinding environment.

Furthermore, we consider the set of essentially free type variables of a type σ under a kinding environment K (denoted as $EFTV(K, \sigma)$) as the smallest set such that, $FTV(\sigma) \subseteq EFTV(K, \sigma)$ and if $\alpha \in EFTV(K, \sigma)$, then $FTV(K(\alpha)) \subseteq EFTV(K, \sigma)$. This reflects the fact that a type variable α is essentially free in σ under a kinding environment K, if α is free in σ or in a restriction in K.

Definition 2.5 Let τ be a monotype, κ a kind, and K a kinding environment. Then we say that τ has kind κ under K (written $K \Vdash \tau :: \kappa$), if $\tau :: \kappa$ can be obtain by applying the following rules:

$$
\begin{aligned}
&K \Vdash \tau :: \mathcal{U} \text{ for all } \tau \text{ well-formed under } K \\
&K \Vdash \alpha :: \{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\} \text{ if } K(\alpha) = \{\{l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots\}\} \\
&K \Vdash \{l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots\} :: \{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\} \\
&\qquad \text{if } \{l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots\} \text{ is well-formed under } K
\end{aligned}
$$

Note that, if $K \Vdash \sigma :: \kappa$, then σ and κ are well-formed under K.

Example 2.6 Let $\tau = \alpha_1 \to \{l_2 : int, l_3 : (\alpha_2 \times \alpha_3)\}$. Then, τ is well-formed under $K_1 = \{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}, \alpha_3 :: \mathcal{U}\}$, because $FTV(\tau) \subseteq dom(K_1)$, but not under $K_2 = \{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}$, because $\alpha_3 \notin dom(K_2)$, and, therefore, $FTV(\tau) \not\subseteq dom(K_2)$. Because τ is well-formed under K_1, we can write $K_1 \Vdash \tau :: \mathcal{U}$.

3 Type assignment

We now define how types are assigned to `EVL` terms. Because we are dealing with polymorphic type schemes, we need to define the notion of *generic instance* for which we first need to discuss well-formed substitutions.

A substitution $S = [\sigma_1/\alpha_1, \ldots, \sigma_n/\alpha_n]$ is *well-formed under a kinding environment* K, if for all $\alpha \in dom(S)$, $S(\alpha)$ is well-formed under K. This reflects the fact that applying a substitution to a type that is well-formed under a kinding environment K, should result in a type that is also well-formed under K. A *kinded substitution* is a pair (K, S) of a kind assignment K and a substitution S that is well-formed under K. This reflects the fact that a substitution S should only be applied to a type that is well-formed under S, such that the resulting type is kinded by K.

Example 3.1 Let $S = [\alpha_2/\alpha_1]$ be a substitution. Then $dom(S) = \{\alpha_1\}$, $S(\alpha_1) = \alpha_2$, and $FTV(\alpha_2) = \{\alpha_2\}$. For the kinding environment $K_1 = \{\alpha_2 :: \kappa\}$, we have that S is well-formed under K_1, since $\alpha_2 \in dom(K_1)$. On the other hand, for the kinding environment $K_2 = \{\alpha_3 :: \kappa\}$, we have that S is not well-formed under K_2, since $\alpha_2 \notin dom(K_2)$.

Definition 3.2 We say that a kinded substitution (K_1, S) respects a kinding environment K_2, if $\forall \alpha \in dom(K_2), K_1 \Vdash S(\alpha) :: S(K_2(\alpha))$.

Example 3.3 Let $K_1 = \{\alpha_1 :: \{\{l_1 : \alpha_2\}\}, \alpha_2 :: \mathcal{U}\}$ and $S = [\{l_1 : int\}/\alpha_1]$. Then, the restricted substitution (K_1, S) respects $K_2 = \{\alpha_1 :: \{\{l_1 : int\}\}\}$, because for $dom(K_2) = \{\alpha_1\}$, we have:

$$\begin{aligned}
&K_1 \Vdash S(\alpha_1) :: S(K_2(\alpha_1))\\
&K_1 \Vdash S(\alpha_1) :: S(\{\{l_1 : int\}\})\\
&K_1 \Vdash S(\alpha_1) :: \{\{l_1 : S(int)\}\}\\
&K_1 \Vdash \{l_1 : int\} :: \{\{l_1 : int\}\}
\end{aligned}$$

Lemma 3.4 *If $FTV(\sigma) \subseteq dom(K)$ and (K_1, S) respects K, then $FTV(S(\sigma)) \subseteq dom(K_1)$.*

Proof. If (K_1, S) respects K, then we know that $\forall \alpha \in dom(K), K_1 \Vdash S(\alpha) :: S(K(\alpha))$. This means that, if $FTV(\sigma) \subseteq dom(K)$, then $\forall \alpha' \in FTV(\sigma), K_1 \Vdash S(\alpha') :: S(K(\alpha'))$. Consequently, both $S(\alpha')$ and $S(K(\alpha'))$ are well formed under K_1, which means that $FTV(S(\alpha')) \subseteq dom(K_1)$ and $FTV(S(K(\alpha'))) \subseteq dom(K_1)$. Therefore, it is now easy to see that $FTV(S(\sigma)) \subseteq dom(K_1)$.

□

Lemma 3.5 *If $K \Vdash \sigma :: \kappa$, and a kinded substitution (K_1, S) respects K, then $K_1 \Vdash S(\sigma) :: S(\kappa)$.*

Proof. The proof follows the definition of $K \Vdash \sigma :: \kappa$ □

Definition 3.6 Let σ_1 be a well-formed type under a kinding environment K. Then, σ_2 is a generic instance of σ_1 under K (denoted as $K \Vdash \sigma_1 \geq \sigma_2$), if $\sigma_1 = \forall \alpha_1 :: \kappa_1^1 \cdots \forall \alpha_n :: \kappa_n^1.\tau_1$, $\sigma_2 = \forall \beta_1 :: \kappa_1^2 \cdots \forall \beta_m :: \kappa_m^2.\tau_2$, and there exists a substitution S such that $dom(S) = \{\alpha_1, \ldots, \alpha_n\}$, $(K \cup \{\beta_1 :: \kappa_1^2, \ldots, \beta_m :: \kappa_m^2\}, S)$ respects $K \cup \{\alpha_1 :: \kappa_1^1, \ldots, \alpha_n :: \kappa_n^1\}$, and $\tau_2 = S(\tau_1)$.

Definition 3.7 Let Γ be a typing environment and τ be a type, both well-formed under a kinding environment K. The closure of τ under Γ and K (denoted as $Cls(K, \Gamma, \tau)$) is a pair $(K', \forall \alpha_1 :: \kappa_1 \cdots \forall \alpha_n :: \kappa_n.\tau)$ such that $K' \cup \{\alpha_1 :: \kappa_1, \ldots \alpha_n :: \kappa_n\} = K$ and $\{\alpha_1, \ldots, \alpha_n\} = EFTV(K, \tau) \setminus EFTV(K, \Gamma)$.

Example 3.8 Let $K = \{\alpha_2 :: \mathcal{U}, \alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}, \alpha_1 :: \{\{l_1 : \alpha_2\}\}\}$, $\Gamma = \{x : \alpha_1\}$, and $\tau = \{l_1 : \alpha_2, l_4 : bool\} \to \{l_2 : int, l_3 : (\alpha_3 \times \alpha_4)\}$. Then $Cls(K, \Gamma, \tau) = (\{\alpha_2 :: \mathcal{U}, \alpha_1 :: \{\{l_1 : \alpha_2\}\}\}, \forall \alpha_3 :: \mathcal{U}.\forall \alpha_4 :: \mathcal{U}.\{l_1 : \alpha_2, l_4 : bool\} \to \{l_2 : int, l_3 : (\alpha_3 \times \alpha_4)\})$, because $K = \{\alpha_2 :: \mathcal{U}, \alpha_1 :: \{\{l_1 : \alpha_2\}\}\} \cup \{\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}$, $EFTV(K, \tau) = \{\alpha_2, \alpha_3, \alpha_4, \alpha_1\}$, $EFTV(K, \Gamma) = \{\alpha_1, \alpha_2\}$, and $EFTV(K, \tau) \setminus EFTV(K, \Gamma) = \{\alpha_2, \alpha_3, \alpha_4, \alpha_1\} \setminus \{\alpha_1, \alpha_2\} = \{\alpha_3, \alpha_4\}$.

The type assignment system for `EVL` is given in Figure 1, and can be seen as both a restriction and an extension of the Ohori type system for record types. Unlike Ohori, we do not deal with variant types in this system, but we have additional language constructors, like conditionals and explicit event definition. We use $K, \Gamma \vdash M : \sigma$ to denote that the `EVL` term M has type σ given the type and kind environments Γ and K, respectively.

Example 3.9 Let $M = \{location = l, fire\_danger = d\}$, $\tau_1 = \{location : \alpha_1, fire\_danger : \alpha_2\}$, $\tau_2 = \forall \alpha_1 :: \mathcal{U}.\forall \alpha_2 :: \mathcal{U}.\alpha_1 \to \alpha_2 \to \tau_1$, $\tau_3 = \{location : l_{type}, fire\_danger : fd_{type}\}$, $\tau_4 = l_{type} \to fd_{type} \to \tau_3$, and $\Gamma = \{\text{“}Porto\text{”} : l_{type}, \text{“}low\text{”} : fd_{type}\}$. A type derivation for M is given in Figure 2.

$$\frac{K \Vdash \Gamma(x) \geq \tau,\ \Gamma \textit{ is well-formed under } K}{K, \Gamma \vdash x : \tau}\ (\mathrm{Var})$$

$$\frac{K, \Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad K, \Gamma \vdash M_2 : \tau_1}{K, \Gamma \vdash M_1\ M_2 : \tau_2}\ (\mathrm{App})$$

$$\frac{K, \Gamma_x \cup \{x : \tau_1\} \vdash M : \tau_2}{K, \Gamma_x \vdash \lambda x.M : \tau_1 \to \tau_2}\ (\mathrm{Abs})$$

$$\frac{K', \Gamma_x \vdash M_1 : \tau' \qquad \mathit{Cls}(K', \Gamma_x, \tau') = (K, \sigma) \qquad K, \Gamma_x \cup \{x : \sigma\} \vdash M_2 : \tau}{K, \Gamma_x \vdash \mathrm{let}\ x = M_1\ \mathrm{in}\ M_2 : \tau}\ (\mathrm{Let})$$

$$\frac{K', \Gamma_x \vdash M_1 : \gamma \qquad \mathit{Cls}(K', \Gamma_x, \gamma) = (K, \sigma) \qquad K, \Gamma_x \cup \{x : \sigma\} \vdash M_2 : \tau}{K, \Gamma_x \vdash \mathrm{letEv}\ x = M_1\ \mathrm{in}\ M_2 : \tau}\ (\mathrm{LetEv})$$

$$\frac{K, \Gamma \vdash M_i : \tau_i, 1 \leq i \leq n}{K, \Gamma \vdash \{l_1 = M_1, \ldots, l_n = M_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}, n \geq 1}\ (\mathrm{Rec})$$

$$\frac{K, \Gamma \vdash M : \tau' \qquad K \Vdash \tau' :: \{\{l : \tau\}\}}{K, \Gamma \vdash M.l : \tau}\ (\mathrm{Sel})$$

$$\frac{K, \Gamma \vdash M_1 : \tau \qquad K, \Gamma \vdash M_2 : \tau' \qquad K \Vdash \tau :: \{\{l : \tau'\}\}}{K, \Gamma \vdash \mathrm{modify}(M_1, l, M_2) : \tau}\ (\mathrm{Modif})$$

$$\frac{K, \Gamma \vdash M_1 : \mathit{bool} \qquad K, \Gamma \vdash M_2 : \tau \qquad K, \Gamma \vdash M_3 : \tau}{K, \Gamma \vdash \mathrm{if}\ M_1\ \mathrm{then}\ M_2\ \mathrm{else}\ M_3 : \tau}\ (\mathrm{Cond})$$

Fig. 1. Type assignment system for `EVL`

Lemma 3.10 *If $K, \Gamma \vdash M : \sigma$ and (K_1, S) respects K, then $K_1, S(\Gamma) \vdash M : S(\sigma)$.*

Proof. By induction on $K, \Gamma \vdash M : \sigma$. □

4 A type inference algorithm for `EVL`

We now adapt Ohori's $\mathit{WK}(K, \Gamma, M)$ inference algorithm to our language. It uses a refinement of Robinson's unification algorithm [27] that considers kind constraints on type variables. We start by discussing *kinded unification* for `EVL` types.

4.1 Kinded unification

A *kinded set of equations* is a pair (K, E), where K is a kinding environment and E is a set of pairs of types (τ_1, τ_2) that are well-formed under K. A *kinded substitution* (K, S) is a unifier of a kinded set of equations (K, E), if every type that appears in E respects K, and $\forall(\tau_1, \tau_2) \in E, S(\tau_1) = S(\tau_2)$ (S satisfies E). A kinded substitution (K_1, S_1) is the *most general unifier* of (K, E) if it is a unifier of (K, E) and if for any other unifier (K_2, S_2) of (K, E) there is some substitution S_3 such that (K_2, S_3) respects K_1 and $S_2 = S_3 \circ S_1$.

The *kinded unification algorithm*, $U(E, K)$, is defined by the transformation rules in Figure 3. Each rule is of the form $(E_1, K_1, S_1) \Rightarrow (E_2, K_2, S_2)$, where E_1, E_2 are sets of pairs of types, K_1, K_2 are kinding environments, and S_1, S_2 are substitutions. After a transformation step, E_2 keeps the set of pairs of types to be unified, K_2

$$\Phi_1 = \dfrac{\dfrac{\dfrac{\dfrac{}{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1, d : \alpha_2\} \ \cup\ \Gamma \vdash l : \alpha_1}\,(\text{Var}) \qquad \dfrac{}{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1, d : \alpha_2\} \ \cup\ \Gamma \vdash d : \alpha_2}\,(\text{Var})}{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1, d : \alpha_2\} \ \cup\ \Gamma \vdash M : \tau_1}\,(\text{Rec})}{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1\} \ \cup\ \Gamma \vdash \lambda d.M : \alpha_2 \to \tau_1}\,(\text{Abs})}{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \Gamma \vdash \lambda l.\lambda d.M : \alpha_1 \to \alpha_2 \to \tau_1}\,(\text{Abs})$$

$$\Phi_2 = \mathit{Cls}(\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \Gamma, \alpha_1 \to \alpha_2 \to \tau_1) = (\{\}, \tau_2)$$

$$\Phi_3 = \dfrac{\dfrac{}{\{\}, \{\mathit{FireDanger} : \tau_2\} \ \cup\ \Gamma \vdash \mathit{FireDanger} : \tau_4}\,(\text{Var}) \qquad \dfrac{}{\{\}, \{\mathit{FireDanger} : \tau_2\} \ \cup\ \Gamma \vdash \text{“}\mathit{Porto}\text{”} : l_{type}}\,(\text{Var})}{\{\}, \{\mathit{FireDanger} : \tau_2\} \ \cup\ \Gamma \vdash \mathit{FireDanger}\ \text{“}\mathit{Porto}\text{”} : fd_{type} \to \tau_3}\,(\text{App})$$

$$\Phi_4 = \dfrac{\Phi_3 \qquad \dfrac{}{\{\}, \{\mathit{FireDanger} : \tau_2\} \ \cup\ \Gamma \vdash \text{“}\mathit{low}\text{”} : fd_{type}}\,(\text{Var})}{\{\}, \{\mathit{FireDanger} : \tau_2\} \ \cup\ \Gamma \vdash \mathit{FireDanger}\ \text{“}\mathit{Porto}\text{”}\ \text{“}\mathit{low}\text{”} : \tau_3}\,(\text{App})$$

$$\dfrac{\Phi_1 \qquad \Phi_2 \qquad \Phi_4}{\{\}, \Gamma \vdash \text{letEv}\ \mathit{FireDanger} = \lambda l.\lambda d.M\ \text{in}\ \mathit{FireDanger}\ \text{“}\mathit{Porto}\text{”}\ \text{“}\mathit{low}\text{”} : \tau_3}\,(\text{LetEv})$$

Fig. 2. Type derivation for $M = \text{letEv}\ \mathit{FireDanger} = \lambda l.\lambda d.M$ in $\mathit{FireDanger}$ “*Porto*” “*low*”

specifies kind constraints to be verified, and S_2 is the substitution resulting from unifying the pairs of types that have been removed from E. Given a kinded set of equations (K_1, E_1) the algorithm $U(E_1, K_1)$ proceeds by applying the transformation rules to $(E_1, K_1, \emptyset)$, until no more rules can be applied, resulting in a triple (E, K, S). If $E = \emptyset$ then it returns the pair (K, S), otherwise it reports failure.

Example 4.1 Let α_1 and α_2 be two type variables and $K = \{\alpha_1 :: \{\{location : \alpha_3\}\},\ \alpha_2 :: \{\{fire\_danger : fd_{type}, location : l_{type}\}\}, \alpha_3 :: \mathcal{U}\}$.

$$\begin{aligned}
&(\{(\alpha_1, \alpha_2)\}, \{(\alpha_1, \{\{location : \alpha_3\}\}), (\alpha_2, \{\{fire\_danger : fd_{type}, location : l_{type}\}\}), (\alpha_3, \mathcal{U})\}, \{\})\\
&\Rightarrow (\{(\alpha_3, l_{type})\}, \{(\alpha_2, \{\{fire\_danger : fd_{type}, location : \alpha_3\}\}), (\alpha_3, \mathcal{U})\}, \{(\alpha_1, \alpha_2)\})\\
&\Rightarrow (\{\}, \{(\alpha_2, \{\{fire\_danger : fd_{type}, location : l_{type}\}\})\}, \{(\alpha_1, \alpha_2), (\alpha_3, l_{type})\})
\end{aligned}$$

The most general unifier between α_1 and α_2 is the kinded substitution $(\{\alpha_2 :: \{\{fire\_danger : fd_{type}, location : l_{type}\}\}\}, [\alpha_2/\alpha_1, l_{type}/\alpha_3])$. Following Ohori's notation, in the kinded unification algorithm we use pairs in the representation of substitutions and kind assignments. Also, note that the unification algorithm in [24] has a kind assignment as an extra parameter, which is used to record the solved kind constraints encountered through the unification process. However, we choose to omit this parameter because its information is only used in the proofs in [24] but not in the unification process itself.

In [24], the correctness and completeness of the kinded unification algorithm was proved, in the sense that it takes any kinded set of equations and computes its most general unifier if one exists and reports failure otherwise.

4.2 Type inference

The *type inference algorithm*, $WK(K, \Gamma, M)$, is defined in Figure 4. Given a kinding environment K, a typing environment Γ, and an `EVL` term M, then $WK(K_1, \Gamma, M) = (K', S, \sigma)$, such that σ is the type of M under the kinding environment K' and typing environment $S(\Gamma)$. It is implicitly assumed that the inference algorithm fails if unification or any of the recursive calls on subterms fails.

Example 4.2 Following Example 3.9, we consider $M = \{location = l, fire\_danger = d\}$, $\tau_1 = \{location : \alpha_1, fire\_danger : \alpha_2\}$, $\tau_2 = \forall\alpha_1 :: \mathcal{U}.\forall\alpha_2 :: \mathcal{U}.\alpha_1 \to \alpha_2 \to \tau_1$, $\tau_3 = \{location : l_{type}, fire\_danger : fd_{type}\}$, and $\tau_4 = l_{type} \to fd_{type} \to \tau_3$, we further consider $\tau_5 = \{location : \alpha_3, fire\_danger : \alpha_4\}$, $\tau_6 = \forall\alpha_3 :: \mathcal{U}.\forall\alpha_4 :: \mathcal{U}.\alpha_3 \to \alpha_4 \to \tau_5$, $\tau_7 = \{location : \alpha_5, fire\_danger : \alpha_6\}$, $S_1 = [l_{type}/\alpha_5] \circ [\alpha_5/\alpha_3, \alpha_6/\alpha_4]$, and $S_2 = [fd_{type}/\alpha_6] \circ S_1 \circ [\alpha_4/\alpha_2] \circ [\alpha_3/\alpha_1]$. A run of the algorithm for letEv $\mathit{FireDanger} = \lambda l.\lambda d.M$ in $\mathit{FireDanger}$ “*Porto*” “*low*” is given in Figure 5.

$$(E \cup \{(\tau,\tau)\}, K, S) \Rightarrow (E, K, S)$$

$$(E \cup \{(\alpha,\tau)\}, K \cup \{(\alpha,\mathcal{U})\}, S) \Rightarrow ([\tau/\alpha]E, [\tau/\alpha]K, [\tau/\alpha]S \cup \{(\alpha,\tau)\})$$

$$(E \cup \{(\tau,\alpha)\}, K \cup \{(\alpha,\mathcal{U})\}, S) \Rightarrow ([\tau/\alpha]E, [\tau/\alpha]K, [\tau/\alpha]S \cup \{(\alpha,\tau)\})$$

$$\begin{aligned}(E \cup \{(\alpha_1,\alpha_2)\}, K \cup \{(\alpha_1, \{\{F_1\}\}), (\alpha_2, \{\{F_2\}\})\}, S) \Rightarrow\ & ([\alpha_2/\alpha_1](E \cup \{(F_1(l), F_2(l)) \mid l \in \mathit{dom}(F_1) \cap \mathit{dom}(F_2)\}),\\ & [\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1](\{\{F_1 \pm F_2\}\}))\},\\ & [\alpha_2/\alpha_1](S) \cup \{(\alpha_1,\alpha_2)\})\end{aligned}$$

$$\begin{aligned}(E \cup \{(\alpha, \{F_2\})\}, K \cup \{(\alpha, \{\{F_1\}\})\}, S) \Rightarrow\ & ([\{F_2\}/\alpha](E \cup \{(F_1(l), F_2(l)) \mid l \in \mathit{dom}(F_1)\}),\\ & [\{F_2\}/\alpha](K),\\ & [\{F_2\}/\alpha](S) \cup \{(\alpha, \{F_2\})\})\\ & \text{if } \mathit{dom}(F_1) \subseteq \mathit{dom}(F_2) \text{ and } \alpha \notin \mathrm{FTV}(\{F_2\})\end{aligned}$$

$$\begin{aligned}(E \cup \{(\{F_2\}, \alpha)\}, K \cup \{(\alpha, \{\{F_1\}\})\}, S) \Rightarrow\ & ([\{F_2\}/\alpha](E \cup \{(F_1(l), F_2(l)) \mid l \in \mathit{dom}(F_1)\}),\\ & [\{F_2\}/\alpha](K),\\ & [\{F_2\}/\alpha](S) \cup \{(\alpha, \{F_2\})\})\\ & \text{if } \mathit{dom}(F_1) \subseteq \mathit{dom}(F_2) \text{ and } \alpha \notin \mathrm{FTV}(\{F_2\})\end{aligned}$$

$$\begin{aligned}(E \cup \{(\{F_1\}, \{F_2\})\}, K, S) \Rightarrow\ & (E \cup \{(F_1(l), F_2(l)) \mid l \in \mathit{dom}(F_1)\}, K, S)\\ & \text{if } \mathit{dom}(F_1) = \mathit{dom}(F_2)\end{aligned}$$

$$(E \cup \{(\tau_1^1 \to \tau_1^2, \tau_2^1 \to \tau_2^2\}, K, S) \Rightarrow (E \cup \{(\tau_1^1, \tau_2^1), (\tau_1^2, \tau_2^2)\}, K, S)$$

Fig. 3. Kinded Unification

4.3 Soundness and completeness of WK

In this section we establish the results from soundness and completeness of our type inference algorithm.

Theorem 4.3 *If $WK(K, \Gamma, M) = (K', S, \tau)$ then (K', S) respects K and there is a derivation in our type system such that $K', S(\Gamma) \vdash M : \tau$.*

Proof. The proof is by induction on the structure of M. □

Theorem 4.4 *If $WK(K, \Gamma, M) = \mathit{fail}$, then there is no (K_0, S_0) and τ_0 such that (K_0, S_0) respects K and $K_0, S_0(\Gamma) \vdash M : \tau_0$.*
If $WK(K, \Gamma, M) = (K', S, \tau)$, then if $K_0, S_0(\Gamma) \vdash M : \tau_0$ for some (K_0, S_0) and τ_0 such that (K_0, S_0) respects K, then there is some S' such that (K_0, S') respects K', $\tau_0 = S'(\tau)$, and $S_0(\Gamma) = S' \circ S(\Gamma)$.

Proof. The proof is by induction on the structure of M. □

5 EVL for Complex Event Processing

In this section we are going to study the application of EVL in the context of Complex Event Processing (CEP). See [15] for a detailed reference on the area. The area of CEP comprises a series of techniques to deal with streams of events such as event processing, detection of patterns and relationships, filtering, transformation and abstraction, amongst others. Because EVL is a higher-order functional language, we are going to explore

$$
\begin{array}{rcl}
\mathit{WK}(K,\Gamma,x) & = & \text{if } x \notin \mathit{dom}(\Gamma) \text{ then } \mathit{fail} \\
 & & \text{else let } \forall\alpha_1 :: \kappa_1 \cdots \forall\alpha_n :: \kappa_n.\tau = \Gamma(x), \\
 & & \qquad\quad S = [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n] \ (\beta_1, \ldots, \beta_n \text{ are fresh}) \\
 & & \qquad \text{in } (K \cup \{\beta_1 :: S(\kappa_1), \ldots, \beta_n :: S(\kappa_n)\}, \mathit{id}, S(\tau)) \\
\mathit{WK}(K,\Gamma,M_1\ M_2) & = & \text{let } (K_1,S_1,\tau_1) = \mathit{WK}(K,\Gamma,M_1) \\
 & & \quad (K_2,S_2,\tau_2) = \mathit{WK}(K_1,S_1(\Gamma),M_2) \\
 & & \quad (K_3,S_3) = \mathit{U}(K_2,\{(S_2(\tau_1),\tau_2 \to \alpha)\}) \ (\alpha \text{ is fresh}) \\
 & & \text{in } (K_3, S_3 \circ S_2 \circ S_1, S_3(\alpha)) \\
\mathit{WK}(K,\Gamma,\lambda x.M) & = & \text{let } (K_1,S_1,\tau) = \mathit{WK}(K \cup \{\alpha :: \mathcal{U}\}, \Gamma \cup \{x : \alpha\}, M) \ (\alpha \text{ fresh}) \\
 & & \text{in } (K_1, S_1, S_1(\alpha) \to \tau) \\
\mathit{WK}(K,\Gamma,\text{let } x = M_1 \text{ in } M_2) & = & \text{let } (K_1,S_1,\tau_1) = \mathit{WK}(K,\Gamma,M_1) \\
 & & \quad (K_1',\sigma) = \mathit{Cls}(K_1,S_1(\Gamma),\tau_1) \\
 & & \quad (K_2,S_2,\tau_2) = \mathit{WK}(K_1',S_1(\Gamma) \cup \{x:\sigma\},M_2) \\
 & & \text{in } (K_2, S_2 \circ S_1, \tau_2) \\
\mathit{WK}(K,\Gamma,\text{letEv } x = M_1 \text{ in } M_2) & = & \text{let } (K_1,S_1,\gamma) = \mathit{WK}(K,\Gamma,M_1) \\
 & & \quad (K_1',\sigma) = \mathit{Cls}(K_1,S_1(\Gamma),\gamma) \\
 & & \quad (K_2,S_2,\tau_2) = \mathit{WK}(K_1',S_1(\Gamma) \cup \{x:\sigma\},M_2) \\
 & & \text{in } (K_2, S_2 \circ S_1, \tau_2) \\
\mathit{WK}(K,\Gamma,\{l_1 = M_1, \ldots, l_n = M_n\}) & = & \text{let } (K_1,S_1,\tau_1) = \mathit{WK}(K,\Gamma,M_1) \\
 & & \quad (K_i,S_i,\tau_i) = \mathit{WK}(K_{i-1},S_{i-1} \circ \cdots \circ S_1(\Gamma),M_i) \ (2 \le i \le n) \\
 & & \text{in } (K_n, S_n \circ \cdots \circ S_2 \circ S_1, \\
 & & \quad \{l_1 : S_n \circ \cdots \circ S_2(\tau_1), \ldots, l_i : S_n \circ \cdots \circ S_{i+1}(\tau_i), \ldots, l_n : \tau_n\}) \\
\mathit{WK}(K,\Gamma,M.l) & = & \text{let } (K_1,S_1,\tau_1) = \mathit{WK}(K,\Gamma,M) \\
 & & \quad (K_2,S_2) = \mathit{U}(K_1 \cup \{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1\}\}\}, \{(\alpha_2,\tau_1)\}) \ (\alpha_1,\alpha_2 \text{ fresh}) \\
 & & \text{in } (K_2, S_2 \circ S_1, S_2(\alpha_1)) \\
\mathit{WK}(K,\Gamma,\text{modify}(M_1,l,M_2)) & = & \text{let } (K_1,S_1,\tau_1) = \mathit{WK}(K,\Gamma,M_1) \\
 & & \quad (K_2,S_2,\tau_2) = \mathit{WK}(K_1,S_1(\Gamma),M_2) \\
 & & \quad (K_3,S_3) = \mathit{U}(K_2 \cup \{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1\}\}\}, \{(\alpha_1,\tau_2),(\alpha_2,S_2(\tau_1))\}) \\
 & & \quad (\alpha_1,\alpha_2 \text{ are fresh}) \\
 & & \text{in } (K_3, S_3 \circ S_2 \circ S_1, S_3(\alpha_2)) \\
\mathit{WK}(K,\Gamma,\text{if } M_1 \text{ then } M_2 \text{ else } M_3) & = & \text{let } (K_1,S_1,\tau_1) = \mathit{WK}(K,\Gamma,M_1) \\
 & & \quad (K_2,S_2) = \mathit{U}(K_1,\{(\tau_1,\mathit{bool})\}) \\
 & & \quad (K_3,S_3,\tau_2) = \mathit{WK}(K_2,S_2 \circ S_1(\Gamma),M_2) \\
 & & \quad (K_4,S_4,\tau_3) = \mathit{WK}(K_3,S_3 \circ S_2 \circ S_1(\Gamma),M_3) \\
 & & \quad (K_5,S_5) = \mathit{U}(K_4,\{(S_4(\tau_2),\tau_3)\}) \\
 & & \text{in } (K_5, S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1, S_5 \circ S_4(\tau_2))
\end{array}
$$

Fig. 4. Type inference algorithm

$$WK(\{\}, \{\}, \text{letEv } \mathit{FireDanger} = \lambda l.\lambda d.M \text{ in } \mathit{FireDanger} \text{ ``}\mathit{Porto}\text{'' ``}\mathit{low}\text{''}) = (\{\}, S_2, \tau_3)$$

$$\begin{array}{l}
WK(\{\}, \{\}, \lambda l.\lambda d.M) = (\{\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, [\alpha_4/\alpha_2] \circ [\alpha_3/\alpha_1], \alpha_3 \to \alpha_4 \to \tau_5) \\
\quad WK(\{\alpha_1 :: \mathcal{U}\}, \{l : \alpha_1\}, \lambda d.M) = (\{\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, [\alpha_4/\alpha_2] \circ [\alpha_3/\alpha_1], \alpha_4 \to \tau_5) \\
\quad\quad WK(\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1, d : \alpha_2\}, M) = (\{\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, [\alpha_4/\alpha_2] \circ [\alpha_3/\alpha_1], \tau_5) \\
\quad\quad\quad WK(\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1, d : \alpha_2\}, l) = (\{\alpha_2 :: \mathcal{U}, \alpha_3 :: \mathcal{U}\}, [\alpha_3/\alpha_1], \alpha_3) \\
\quad\quad\quad WK(\{\alpha_2 :: \mathcal{U}, \alpha_3 :: \mathcal{U}\}, \{l : \alpha_3, d : \alpha_2\}, d) = (\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, [\alpha_4/\alpha_2], \alpha_4)
\end{array}$$

$$\mathit{Cls}(\{\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, \{\}, \alpha_3 \to \alpha_4 \to \tau_5) = (\{\}, \tau_6)$$

$$\begin{array}{l}
WK(\{\}, \{\mathit{FireDanger} : \tau_6\}, \mathit{FireDanger} \text{ ``}\mathit{Porto}\text{'' ``}\mathit{low}\text{''}) = (\{\}, [\mathit{fd}_{type}/\alpha_6] \circ S_1, \tau_3) \\
\quad WK(\{\}, \{\mathit{FireDanger} : \tau_6\}, \mathit{FireDanger} \text{ ``}\mathit{Porto}\text{''}) = (\{\}, S_1, \alpha_6 \to \{\mathit{location} : l_{type}, \mathit{fire\_danger} : \alpha_6\}) \\
\quad\quad WK(\{\}, \{\mathit{FireDanger} : \tau_6\}, \mathit{FireDanger}) = (\{\}, [\alpha_5/\alpha_3, \alpha_6/\alpha_4], \alpha_5 \to \alpha_6 \to \tau_7) \\
\quad\quad WK(\{\}, \{\mathit{FireDanger} : \tau_6\}, \text{``}\mathit{Porto}\text{''}) = (\{\}, id, l_{type}) \\
\quad\quad U(\{\}, (\alpha_5, l_{type})) = (\{\}, [l_{type}/\alpha_5]) \\
\quad WK(\{\}, \{\mathit{FireDanger} : \tau_6\}, \text{``}\mathit{low}\text{''}) = (\{\}, id, \mathit{fd}_{type}) \\
\quad U(\{\}, (\alpha_6, \mathit{fd}_{type})) = (\{\}, [\mathit{fd}_{type}/\alpha_6])
\end{array}$$

Fig. 5. Type inference run for letEv $\mathit{FireDanger} = \lambda l.\lambda d.M$ in $\mathit{FireDanger}$ "*Porto*" "*low*"

the higher-order capabilities, to define higher-order parameterized functions that deal with the usual CEP techniques.

5.1 Event processing

The canonical model [10,15] for event processing is based on a producer-consumer model: an event processing agent (EPA) takes events from event producers and distributes them among event consumers. This process often involves filtering or translating. Filtering may happen because not every event will be of interest or available to every consumer: in some cases access control policies might be in place that restrict what events consumers might receive. Translating events allows us to change, add or remove information from the agents based on particular consumers. The processing of events can be done in a one-event in/one-event out form, but it is also possible to have event processing agents that process a collection of events as a whole or that produce a set of events as result: for example, an incoming event may be split into multiple events, each containing a subset of the information from the original event.

EVL can be used to program event processing agents. It is able to process raw events produced by some event processing system and generate derived events as a result. These derived events can then be passed on to an event consumer.

Principal types allow us to identify event processing agents. An event processing agent is any function whose principal types is of the form $\forall\alpha_1 :: \kappa_1 \cdots \forall\alpha_n :: \kappa_n.\gamma$. In the following section we will explore the different types of event processing agents.

5.1.1 Types of event processing agents

Event processing agents are classified according to the actions that they perform to process incoming events:

- *Filter agents* - Filter agents take an incoming event object and apply a test to decide whether to discard it or whether to pass it on for processing by subsequent agents. The test is usually stateless, i.e. based solely on the content of the event instance.
- *Transformation agents* - Transformation agents modify the content of the event objects that they receive. These agents can be further classified based on the cardinality of their inputs and outputs (translate, split, aggregate or compose agents)
- *Pattern Detect agents* - Pattern Detect agents take collections of incoming event objects and examine them to see if they can spot the occurrence of particular patterns.

We are now going to look into the different types of event processing agents in a little more depth. All of the definitions that we are going to present can be found in [15].

Definition 5.1 [Filter event processing agent] A *filter agent* is an event processing agent that performs filtering only, so it does not transform the input event.

Example 5.2 This example represents an event processing agent that uses a higher-order filter function *filter* (to be defined later) to filter events according to their location.

```
let p x = (x.location == "Porto") in λx.(filter p x)
```

Definition 5.3 [Transformation event processing agent] A Transformation event processing agent is an event processing agent that includes a derivation step, and optionally also a filtering step.

Transformation event processing agents can be either stateless (if events are processed without taking into account preceding or following events) or stateful (if the way events are processed is influenced by preceding or following events). In the former case, events are processed individually. In the latter, the way events are processed can depend on preceding or succeeding events. Transformation events can be further classified as translate, split, aggregate or compose agents. In the following we describe some transformation events that we are going to focus on.

Definition 5.4 [Translate event processing agent] Translate event processing agents can be used to convert events from one type to another, or to add, remove, or modify the values of an event's attributes.

At the moment EVL does not allow us to add or remove attributes. One can create new events based on attributes from incoming events as well as modify the value of existing attributes. We follow Ohori's treatment of record types, therefore we do not consider operations that extend a record with a new field or that remove an existing field from a record. This is a limitation of our current calculus, and we intend to improve on this in the future.

Example 5.5 This example represents an event processing agent that converts the temperature field of an event from degrees Fahrenheit to degrees Celsius.

```
farToCel x = modify(x, temperature, (x.temperature −32)/1.8)
```

Definition 5.6 [Aggregate event processing agent] Aggregate agents take a stream of incoming events and produce an output event that is a map of the incoming events.

Example 5.7 This example represents an event processing agent that receives two events, x and y, and outputs event y with its precipitation level updated with the average of the two.

```
avg x y = modify(y, precipitation, (x.precipitation + y.precipitation)/2)
```

Definition 5.8 [Compose event processing agent] Compose agents take two streams of incoming events and process them to produce a single output stream of events.

Example 5.9 This example represents an event processing agent that composes the partial weather information that is provided by two different sensors. One of the sensors outputs event x, which contains information about the temperature and wind velocity, and the other sensor outputs event y, which contains information about the humidity and precipitation levels. This event processing agent outputs an instance of *WeatherInfo* with the complete weather information.

```
composeInfo x y = WeatherInfo x.temperature x.wind y.humidity y.precipitation
```

Definition 5.10 [Pattern Detect event processing agent] A Pattern Detect event processing agent is an event processing agent that performs a pattern matching function on one or more input streams. It emits one or more derived events if it detects an occurrence of the specified pattern in the input events.

Example 5.11 The *check* function in Example 2.3 is an event processing agent that generates the appropriate *FireDanger* event by detecting its corresponding fire weather information.

```
check x = if (x.temperature > 29 and x.wind > 32
              and x.humidity < 20 and x.precipitation < 50)
          then FireDanger x.location "high"
          else FireDanger x.location "low"
```

An area where pattern-detect agents are quite relevant is in publish-subscribe systems, where consumers are allowed to subscribe to selective events by specifying filters using a subscription language. The filters define constraints, usually in the form of name-value pairs of properties and basic comparison operators, which identify valid events. Constraints can be logically combined to form what are called complex subscription patterns [16]. CEP systems extend the functionality of traditional publish-subscribe systems by increasing the expressive power of the subscription language to consider complex event patterns that involve the occurrence of multiple, related events [12].

5.1.2 A higher-order library for CEP

Since EVL is a higher-order language, we use higher-order functions that allow us to deal with a sequence of events (represented as a list of events). We now provide some of these useful higher-order functions, which are naturally implemented in a higher-order functional language.

- `filter` is a function that filters the events in the sequence according to some filtering expression:

```
filter p list = if list.empty then list
                else if (p list.head) then (cons list.head (filter p list.tail))
                     else filter p list.tail
```

- `transform` is a function that applies a transformation to all of the events in the sequence:

```
transform f list = if list.empty then list
                   else (cons (f list.head) (transform f list.tail))
```

- `aggregater` is a function that produces some output value by aggregating by right association the events of the sequence according to some binary aggregating function:

```
aggregater f z list = if list.empty then z
                      else f list.head (aggregater f z list.tail)
```

- `aggregatel` is very similar to `aggregater` but it aggregates the events by left association:

```
aggregatel f z list = if list.empty then z
                      else aggregatel f (f z list.head) list.tail
```

We use the function names `transform` and `aggregateX`, instead of the usual functional programming names `map` and `foldX`, because of the particular context of CEP.

5.2 Event types

When dealing with event processing applications, many events will have a similar structure and a similar meaning. Consider a temperature sensor: all of the events produced by it have the same kind of information, such as temperature reading, timestamp and maybe location, but with possibly different values. This means that instead of defining the structure of each event individually, we can specify the structure of an entire class of events [15]. This relationship was formally defined in [1] as that between *Generic* and *Specific* events. EVL is based on the typed language that was defined in [1], but it extends it by allowing explicit subtyping between record types according to [24].

The definition of a generic event may contain references to other events when there is a semantic relationship between them. These relationships can be separated into four types [15] that we are now going to discuss.

Membership

A generic event ge_1 is said to be a member of another generic event ge_2 if the instances of ge_1 are included in the instances of ge_2. This notion can easily be checked in EVL through the explicit subtyping relation between record types.

Generalization

The generalization relation indicates that an event is a generalization of another event. In the type theory of EVL, this relation is given by the generalization relation *Cls*.

Specialization

The specialization relation indicates that an event is a specialization of another event. In the type theory of EVL, this relation is given by the type instantiation relation.

Retraction

A retraction event relationship is a property of an event referencing a second event. It indicates that the second event is a logical reversal of the event type that references it. For example, an event that starts a fire alert and the event that stops it. This is a notion that is also present in access control systems with obligations. In [1], this is defined by a closing function that describes how events are linked to subsequent events in history.

That is, which are the generic events in time that are closed by a particular generic event provided that some time constraints are satisfied and following a specific strategy. These functions are assumed to be defined for each system and are used to extract intervals from a given history. One of the motivations to develop `EVL` was to provide a simple language to program such functions.

5.3 A comprehensive example

We now give an example that illustrates several features described in this section.

Example 5.12 Consider a sequence of events produced by sensors distributed across some number of locations. The events produced by a particular sensor contains information about the weather conditions at that sensor's location. More specifically, it contains information about the temperature (in degrees Celsius), the humidity level (as a percentage), the wind speed (in km/h) and the amount of precipitation (in mm), as well as information about its location. Now, consider an EPA that infers the fire danger of a particular location based on a given sequence of events produced by an arbitrary number of these sensors. This can be done with varying degrees of accuracy, but this is not the subject of this paper, so let us consider a simple algorithm based on the following three steps:

(i) Filtering the events according to the specified location;

(ii) Aggregating the events according to the latest values of temperature, humidity and wind speed, and by the mean precipitation;

(iii) Producing an event that indicates if there is fire danger in that particular location considering the values obtained in the previous step and comparing them to their threshold levels.

We now provide an implementation of this algorithm in EVL:

```
letEv FireDanger l d = {location = l, fire_danger = d} in
let p x = (x.location == "Porto") in
let f x y = (x.fst + 1, modify(y, precipitation,
                              (x.snd.precipitation + y.precipitation)/x.fst)) in
let check x = if (x.temperature > 29 and x.wind > 32
                  and x.humidity < 20 and x.precipitation < 50)
              then FireDanger x.location "high"
              else FireDanger x.location "low" in
λx.(check (aggregatel f (1, {precipitation = 0}) (filter p x)).snd)
```

6 Related Work

Alternative type systems to deal with records have been presented in the literature using row variables [28], which are variables ranging over finite sets of field types. One of the most flexible systems using row variables [25] allows for powerful operations on records, such as extending a record with a new field or removing an existing field from a record. Record extension is also available in other systems [22,19,9], as well as record concatenation operations [20,26,29], however adding these operations results in complications in the typing process. By following Ohori's approach we obtain a sound and complete efficient type system supporting the basic operations for dealing with records. Nevertheless, regarding the applicability of our language in the context of CEP, the integration of more flexible record operations in our language is an aspect to be further investigated.

When it comes to processing flows of information, there are two main models leading the research done in this area: the data stream processing model [4] (that looks at streams of data coming from different sources to produce new data streams as output); and the complex event processing model [23] (that looks at events happening, which are then filtered and combined to produce new events). In [12], several information processing systems were surveyed, which showed a gap between data processing languages and event detection languages, and the need to define a minimal set of language constructors to combine both features in the same language. We believe that EVL is a good candidate to explore the gap between these two models.

Following the complex event processing model, one of the key features is the ability to derive complex events (composite) from lower-level events and several special purpose Event Query Languages (EQLs) have been proposed for that [14]. Complex event queries over real-time streams of RFID readings have been dealt with in [30] yet again using a query language. The TESLA language [11] supports content-based event filtering as well as been able to establish temporal relations on events, while providing a formal semantics based on temporal logic. The lack of a simple denotational semantics is a common criticism of CEP query languages [31,3,17], with several languages not guaranteeing important language features, such as orthogonality, as well as an overlapping of definitions that make reasoning about these languages that much harder. Recently, a formal framework based on a complex event logic (CEL) was proposed [18], with the purpose of "giving a rigorous and efficient framework to CEP". The authors define well-formed and safe formulas, as syntactic restrictions that characterize semantic properties, and argue that only well-formed formulas should be considered and that users should understand that all variables in a formula must be correctly defined. This notion of well-formed formulas and correctly defined variables is naturally guaranteed in a typed language like `EVL`. Therefore we believe that `EVL` can be used to provide formal semantics to CEP systems.

In the context of access control systems, the *Obligation Specification Language* (OSL) defined in [21], presents a language for events to monitor and reason about data usage requirements. The paper defines the *refinesEv* instance relation between events, which is based on a subset relation on labels, as is the case for the instance relation in [2]. The instance relation in [1] was defined by implicit subtyping on records but more generally using variable instantiation. In this paper we further generalise the notion of instance relation and define it formally using kinded instantiation.

Still in the context of access control, Barker et al [7] have given a representation of events as finite sets of ground 2-place facts (atoms) that describe an event, uniquely identified by $e_i, i \in \mathbb{N}$, and which includes three necessary facts: $\mathit{happens}(e_i, t_j)$, $\mathit{act}(e_i, a_l)$ and $\mathit{agent}(e_i, u_n)$, and n non-necessary facts. This representation is claimed to be more flexible than a term-based representation with a fixed set of attributes. The language in this paper is flexible enough to encode the event representation in [7]. Furthermore, the typing system allows us to guarantee any necessary facts by means of the typing information.

7 Conclusions and Future Work

In this paper we present `EVL`, a typed higher-order functional language for events, with a typing system based on Ohori's record calculus, as well as a sound and complete type inference algorithm. We explore the expressiveness of our language by showing its application in the context of CEP. This is a starting point to an area of research exploring the well-studied properties of higher-order functional typed languages and their ability to reason with dynamical properties of systems, while applying it to the areas of obligation models for access control and complex event processing (CEP). We believe the language defined in this paper proved to be better equipped to deal with generic events when compared to traditional relational languages used in the CEP area.

With respect to future work, events in our language are represented by records of the form $\{l_1 = v_1, \ldots, l_n = v_n\}$, with appropriate constructors for creating, accessing and modifying records. Additionally one could consider more powerful operations on records, such as extending a record with a new field or removing an existing field from a record, which are not part of `EVL` but could prove useful in both CEP and in the treatment of obligations in the context of access control models.

We would also like to fully exploit `EVL`'s capabilities in the context of the CBACO metamodel [2]. Furthermore, we would like to explore extensions of `EVL` with pattern matching, which is a powerful mechanism for decomposing and processing data. The ability to detect patterns is a key notion in most CEP systems, therefore adding matching primitives to `EVL` would greatly improve its capability with respect to pattern detection.

Acknowledgment

This work was partially funded by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020 and by Project "Safe Cities", ref. POCI-01-0247-FEDER-041435, financed by Fundo Europeu de Desenvolvimento Regional (FEDER), through COMPETE 2020 and Portugal 2020.

References

[1] Alves, S., S. Broda and M. Fernández, *A typed language for events*, in: M. Falaschi, editor, *Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2015, Siena, Italy, July 13-15, 2015. Revised Selected Papers*, Lecture Notes in Computer Science **9527** (2015), pp. 107–123.
URL `https://doi.org/10.1007/978-3-319-27436-2_7`

[2] Alves, S., A. Degtyarev and M. Fernández, *Access Control and Obligations in the Category-Based Metamodel: A Rewrite-Based Semantics*, in: *Proceedings of LOPSTR'14*, LNCS **8981**, Springer, 2015 pp. 148–163.

[3] Artikis, A., A. Margara, M. Ugarte, S. Vansummeren and M. Weidlich, *Complex event recognition languages: Tutorial*, in: *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, DEBS '17 (2017), p. 7–10.
URL `https://doi.org/10.1145/3093742.3095106`

[4] Babcock, B., S. Babu, M. Datar, R. Motwani and J. Widom, *Models and issues in data stream systems*, in: *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02 (2002), p. 1–16.
URL `https://doi.org/10.1145/543613.543615`

[5] Barendregt, H. P., "The lambda calculus - its syntax and semantics," Studies in logic and the foundations of mathematics **103**, North-Holland, 1985.

[6] Barga, R. S., J. Goldstein, M. H. Ali and M. Hong, *Consistent streaming through time: A vision for event stream processing*, in: *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings* (2007), pp. 363–374.
URL http://cidrdb.org/cidr2007/papers/cidr07p42.pdf

[7] Barker, S., M. J. Sergot and D. Wijesekera, *Status-Based Access Control*, ACM Transactions on Information and System Security **12** (2008), pp. 1:1–1:47.

[8] Bertino, E., P. A. Bonatti and E. Ferrari, *TRBAC: A Temporal Role-based Access Control Model*, ACM Transactions on Information and System Security **4** (2001), pp. 191–233.

[9] Cardelli, L. and J. C. Mitchell, *Operations on records*, in: M. Main, A. Melton, M. Mislove and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics* (1990), pp. 22–52.

[10] Chandy, M. K., O. Etzion and R. von Ammon, *The event processing manifesto*, in: K. M. Chandy, O. Etzion and R. von Ammon, editors, *Event Processing*, number 10201 in Dagstuhl Seminar Proceedings (2011).
URL http://drops.dagstuhl.de/opus/volltexte/2011/2985

[11] Cugola, G. and A. Margara, *Tesla: A formally defined event specification language*, in: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10 (2010), p. 50–61.
URL https://doi.org/10.1145/1827418.1827427

[12] Cugola, G. and A. Margara, *Processing flows of information: From data stream to complex event processing*, ACM Comput. Surv. **44** (2012).
URL https://doi.org/10.1145/2187671.2187677

[13] Damas, L. and R. Milner, *Principal type-schemes for functional programs*, in: R. A. DeMillo, editor, *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982* (1982), pp. 207–212.
URL https://doi.org/10.1145/582153.582176

[14] Eckert, M., F. Bry, S. Brodt, O. Poppe and S. Hausmann, "A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed," Springer Berlin Heidelberg, Berlin, Heidelberg, 2011 pp. 47–70.
URL https://doi.org/10.1007/978-3-642-19724-6_3

[15] Etzion, O. and P. Niblett, "Event Processing in Action," Manning Publications Co., USA, 2010, 1st edition.

[16] Eugster, P. T., P. A. Felber, R. Guerraoui and A.-M. Kermarrec, *The many faces of publish/subscribe*, ACM Comput. Surv. **35** (2003), p. 114–131.
URL https://doi.org/10.1145/857076.857078

[17] Galton, A. and J. C. Augusto, *Two approaches to event definition*, in: A. Hameurlain, R. Cicchetti and R. Traunmüller, editors, *Database and Expert Systems Applications* (2002), pp. 547–556.

[18] Grez, A., C. Riveros and M. Ugarte, *A Formal Framework for Complex Event Processing*, in: P. Barcelo and M. Calautti, editors, *22nd International Conference on Database Theory (ICDT 2019)*, Leibniz International Proceedings in Informatics (LIPIcs) **127** (2019), pp. 5:1–5:18.
URL http://drops.dagstuhl.de/opus/volltexte/2019/10307

[19] Harper, R. and J. C. Mitchell, *On the type structure of standard ML*, ACM Trans. Program. Lang. Syst. **15** (1993), pp. 211–252.
URL https://doi.org/10.1145/169701.169696

[20] Harper, R. and B. C. Pierce, *A record calculus based on symmetric concatenation*, in: D. S. Wise, editor, *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991* (1991), pp. 131–142.
URL https://doi.org/10.1145/99583.99603

[21] Hilty, M., A. Pretschner, D. A. Basin, C. Schaefer and T. Walter, *A Policy Language for Distributed Usage Control*, in: *Proceedings of ESORICS'07*, 2007, pp. 531–546.

[22] Jategaonkar, L. A. and J. C. Mitchell, *Type inference with extended pattern matching and subtypes*, Fundam. Inf. **19** (1993), p. 127–165.

[23] Luckham, D., "The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems," Addison-Wesley, Boston, MA, 2002.

[24] Ohori, *A Polymorphic Record Calculus and its Compilation*, ACM Transactions on Programming Languages and Systems **17** (1995), pp. 844–895.

[25] Rémy, D., *Efficient representation of extensible records*, in: *Proceedings of the 1992 workshop on ML and its Applications*, San Francisco, USA, 1992, p. 12.

[26] Rémy, D., *Typing record concatenation for free*, in: R. Sethi, editor, *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992* (1992), pp. 166–176.
URL https://doi.org/10.1145/143165.143202

[27] Robinson, J. A., *A machine-oriented logic based on the resolution principle*, Journal of the Association for Computing Machinery (ACM) **12** (1965), pp. 23–41.

[28] Wand, M., *Complete type inference for simple objects*, in: *Proceedings of the Symposium on Logic in Computer Science (LICS'87), Ithaca, New York, USA, June 22-25, 1987* (1987), pp. 37–44.
URL http://www.ccs.neu.edu/home/wand/papers/wand-lics-87.pdf

[29] Wand, M., *Type inference for record concatenation and multiple inheritance*, in: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989* (1989), pp. 92–97.
URL https://doi.org/10.1109/LICS.1989.39162

[30] Wu, E., Y. Diao and S. Rizvi, *High-performance complex event processing over streams*, in: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06 (2006), p. 407–418.
URL https://doi.org/10.1145/1142473.1142520

[31] Zimmer, D. and R. Unland, *On the semantics of complex events in active database management systems*, in: *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337)*, 1999, pp. 392–399.

Safety of a Smart Classes-Used Regression Test Selection Algorithm

Susannah Mansky[1,2]

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, USA

Elsa L. Gunter[3]

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, USA

Abstract

Regression Test Selection (RTS) algorithms select which tests to rerun on revised code, reducing the time required to check for newly introduced errors. An RTS algorithm is considered safe if and only if all deselected tests would have unchanged results. In this paper, we present a formal proof of safety of an RTS algorithm based on that used by Ekstazi [3], a Java library for regression testing. Ekstazi's algorithm adds print statements to JVM code in order to collect the names of classes used by a test during its execution on a program. When the program is changed, tests are only rerun if a class they used changed. The main insight in their algorithm is that not all uses of classes must be noted, as many necessarily require previous uses, such as when using an object previously created. The algorithm we formally define and prove safe here uses an instrumented semantics to collect touched classes in an even smaller set of locations. We identify problems with Ekstazi's current collection location set that make it not safe, then present a modified set that will make it equivalent to our safe set. The theorems given in this paper have been formalized in the theorem prover Isabelle over JinjaDCI [7], a semantics for a subset of Java and JVM including dynamic class initialization and static field and methods. We instrumented JinjaDCI's JVM semantics by giving a general definition for Collection Semantics, small-step semantics instrumented to collect information during execution. We also give a formal general definition of RTS algorithms, including a definition of safety.

Keywords: interactive theorem proving, regression test selection, small-step semantics, Java

1 Introduction

Testing is a crucial part of writing code. When writing programs it is important to run tests that demonstrate that the behavior of those programs is as expected and documented. When a program is modified, its tests are rerun to make sure changes have not introduced new bugs. This rerunning of tests is called regression testing.

In practice, a body of code can be large and have a huge number of tests written over it. Rerunning every single test can in some cases take hours or days, making it impractical to run them all after every small change. However, most changes will only even possibly affect a small number of tests. Thus, algorithms and methods

[1] This material is based upon work supported in part by the National Science Foundation under Grants CCF-1439957 and CCF 13-18191. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. We would like to thank Milos Gligoric and Darko Marinov for providing the motivating problem and for their insight into the importance of initialization timing. We are also grateful to the reviewers for their time, suggestions, and comments.

[2] Email: sjohnsn2@illinois.edu

[3] Email: egunter@illinois.edu

have been developed to select which tests to rerun after code changes. The process of selecting which tests to run (alternatively, deselecting tests that should not be affected) is called Regression Test Selection (RTS). An RTS algorithm is *safe* if it only deselects tests whose previous results could be reproduced under the modified program.

In Java, all code is written inside methods in classes. Thus one approach to RTS is for Java test suites to determine which classes each test uses ("touches"), then only rerun a test if one of its touched classes has been changed. Ekstazi [3], a Java library for regression testing, uses this approach in its RTS algorithm. Touched classes can be derived in a number of different ways, but Ekstazi's algorithm collects this information dynamically via instrumentation of the code. The naive way to do this is to instrument every use of every class. However, Ekstazi uses a smarter approach, recognizing that many of these collections are likely redundant. For example, fetching a field of an object requires the object to have been created previously, meaning that a `new` has already been run on its class, so `getfield` does not need to trigger class collection. Their approach uses these insights to reduce the number of collection points.

In this paper, we present a formal proof of safety for an RTS algorithm similar to that used by Ekstazi. Our algorithm is designed for minimal collection. We then prove its safety and demonstrate how Ekstazi's algorithm can be modified to be equivalent and thus safe (and showing why the previous set was not safe). The definition of this algorithm is given via instrumentation of JinjaDCI's JVM semantics. These semantics are a model of a subset of the JVM written in the theorem prover Isabelle and are largely described in [7]. The instrumentation is created via a general definition we have given for what we call Collection Semantics that allows information-collection instrumentation of small-step semantics via the input of such semantics and a collection function.

We have also given a general definition of RTS algorithms that includes a definition of safety. Its axioms, including safety, further guarantee that repeated deslection of a test can safely be based on its output under a previous version of the program.[4]

Section 2 gives an overview of Regression Test Selection (RTS), Ekstazi, and JinjaDCI. Section 3 gives details of a few different class collection functions for use in an RTS algorithm using the classes-touched method to select tests. Section 4 describes Collection Semantics, our approach to instrumenting existing semantics with collection functions, and uses this approach to instrument JinjaDCI's JVM byte code with the class collection functions previously described. (These instrumentations are instantiations of the Collection Semantics definition.) Section 5 gives a general Isabelle definition for RTS algorithms, including a formal definition of safety, then combines this with the Collection Semantics definition to define collection-based RTS algorithms. The instantiations of Collection Semantics describing instrumented JinjaDCI JVM semantics are extended into collection-based RTS algorithm instantiations using this combined definition. Section 6 uses the general RTS definition and the instrumented semantics to give proofs of safety of using the defined class collection functions as a basis for classes-touched RTS algorithms. Finally, Sections 7 and 8 discuss some related work, recap, and suggest future directions.

The Isabelle development for the work presented here can be found online at `https://github.com/susannahej/rts`.

2 Background

We will first introduce the relevant topics and tools used in this paper.

2.1 RTS Algorithms and Ekstazi

In industry, many code bases are quite large, as are the test suites associated with them. These test suites can take on the order of days to run in full. Regression test selection (RTS) is the process of choosing which tests to rerun after changes have been made to a code base in order to decrease the time retesting takes. A test that is not run is said to be *deselected*. An RTS algorithm is called *safe* if it only deselects tests whose results would be unchanged.

Ekstazi [3] is a Java library for regression testing that employs an RTS algorithm at the level of JVM bytecode based on the classes that are used or referenced by each test. We call these the classes *touched* by a test. When a test is run, the names of the classes it touches during its execution are collected. Then, when changes are made to the code base, a test is only rerun if one or more of the classes it touched in its previous run have been modified. Thus if, for example, only a couple of modifications are made to non-core classes in a large code base, generally very few tests need to be rerun.

Ekstazi's algorithm attempts to collect the necessary classes as infrequently as possible (i.e., with as few collection triggers as possible), which leaves room for possibly under-collecting. To prove this algorithm's

[4] This is an important feature for an RTS algorithm to have, as deselected tests are, by design, not rerun, so do not provide new outputs until they are reselected.

safety formally requires a semantics of the JVM that includes sufficient features to make the formal proof as convincing as possible. Static instructions and dynamic class initialization in particular are places where a class collection algorithm can be subtly incorrect. In Section 2.2 we will go into more detail about a semantics of the JVM that includes both of these features.

2.2 Isabelle and JinjaDCI

Isabelle allows us to model languages and programs written in them. It further allows the proof of many useful properties of those languages and programs, such as how the information collected during the execution of a program with an instrumented semantics relates to that program.

Jinja [4] is a framework developed in the theorem prover Isabelle whose purpose is to give a formal semantics for a subset of both Java and JVM bytecode in a unified way. The authors wrote the semantics in Isabelle because this allowed them to write definitions and proofs based on their semantics. These proofs include type safety, equivalence of their big- and small-step Java semantics, and the correctness of a compiler from their Java semantics to their JVM semantics.

The two extensions of Jinja are JinjaThreads [6] and JinjaDCI [7]; the former extends Jinja with threads, while the latter extends it with static fields and methods and dynamic class initialization. The features of the latter are crucial uses of classes in Java, with behavior that differs from the other uses described by the original Jinja. For these reasons, these features are important to include in any proof of safety of Ekstazi-like touched-class-collection RTS algorithms like those discussed in this paper. On the other hand, threads should not create any complications with the safety of these algorithms as long as the synchronization of initialization is correct (as further justified in Section 6.4). Thus, we have chosen JinjaDCI as our semantic model for the proofs presented here.

The value of using a system like Isabelle to build a semantic framework is precisely the ability to prove these kinds of results, which is why we chose to use this framework: in order to prove that a test behaves the same on two different programs, we need both a definition of the language and a framework that allows meta-reasoning at this level.

In Jinja's JVM instruction names are capitalized and take appropriate arguments. For example, `new` is written as `New` and takes one argument: the name of the class to be instantiated, C; `getfield` is written as `Getfield` and takes two arguments: the name of the field whose value is being fetched, F, and the name of the class that defines that field in the object being passed, C.

2.2.1 Dynamic Class Initialization

In the JVM, class initialization methods are called dynamically. Rather than initializing classes up front, Java waits until the the class is actually used. When a class C is used, its initialization status is checked. If it is uninitialized, the class initialization procedure is run on it.

Initialization status is checked when a class or one of its subclasses is used. In particular, in the supported subset of JVM instructions, if a class C is not initialized, the class initialization procedure is called when:

- an instruction among `new`, `getstatic`, `putstatic`, or `invokestatic` references C,
- one of C's subclasses is being initialized, or
- at startup of the JVM, if C is the designated initial class.

In the full JVM, the procedure is also called at the invocation of certain bootstrap and reflective methods, and on certain interfaces when an instantiation is initialized.

The first step of the *class initialization procedure* [5] (called on a class C) is to check C's initialization status. If the status is `Error`, then an error is returned. Otherwise, if C needs to be initialized (and is not already being initialized), its superclass is checked and the procedure is run on it as necessary. Then, if that procedure returns normally, C's initialization method is run and its initialization status set to `Done`. If C's or any of its superclasses' methods throw an uncaught exception, then C's initialization status is set to `Error` and its initialization method is not run. [6] No class initialization methods are run until either a superclass is checked that is already initialized or the class `Object` (which has no superclass) is checked. At that point, the initialization methods for all the uninitialized superclasses found along the way are run from top to bottom, ending with C's.

Note how the class initialization *procedure* differs from a class's initialization method: the former is a general series of steps taken to determine the initialization status of a class and its superclasses, and which then calls

[5] The procedure as described here is in brief and assumes a semantics without threads, as in Jinja and JinjaDCI. More precise details of this procedure are given in the JVM specification [5] and the JinjaDCI paper [7].

[6] The thrown exception is returned, and any further attempts to initialize this class will result in an error during the initialization status check as noted in the first step.

the initialization methods of the appropriate classes. The latter is an actual method defined implicitly or explicitly by the definition of a class, and which is invoked in the course of the class initialization procedure.

Note that if a class has the initialization procedure called on it during the execution of a program (*initialization-called* classes), then that class was touched during that execution. (This set includes but is not necessarily limited to those classes actually initialized. A class may call the initialization procedure but fail to run its initialization method if one of its superclasses' initialization methods returns an error.)

Since the RTS algorithms described in this paper function by collecting the names of classes touched by a test (i.e., those classes whose definitions could affect the monitored run), the initialization-called classes are a good portion of the classes that will be collected. In particular, other than these, it turns out that the only other classes touched in a run are those classes that call static methods or fields defined by one of their superclasses. These latter can easily be collected at the time of such calls.

Collecting the initialization-called classes can be done either at the beginning of the initialization procedure or by collecting at points where this procedure might be called (i.e., at initialization checks). The former requires instrumenting the actual semantics of the JVM, because the initialization procedure is called dynamically rather than through instructions in the code. We have taken this approach in our algorithms. The latter is possible to do by adding instructions to the test code and is Ekstazi's approach.

2.2.2 JinjaDCI's JVM

The class-collecting algorithms described in this paper are performed via an instrumentation of JinjaDCI's JVM semantics. The uninstrumented semantics are described in more detail in [7], but the following pieces are a brief summary of those most relevant to the instrumentation.

In JinjaDCI, as in Jinja, a program is a list of class definitions. A JinjaDCI JVM state σ is made up of a heap, a static heap (storing information about static fields and classes' initialization statuses), a frame stack (one frame per active method), and an optional exception. JinjaDCI JVM's execution function `exec` applied to a program P and a state σ will return a next state as long as σ has no exception and its frame stack is non-empty. A frame includes, among other things, an initialization call status flag indicating its current state relative to the initialization procedure. This flag can be built by one of the four constructors `Calling`, `Called`, `Throwing`, and `No_ics`. These are used as follows:

As described in Section 2.2.1, the procedure starts by creating a list of classes to be initialized including the triggering class C and its uninitialized superclasses. During this part of the procedure, the frame that called it has its initialization call status flag set to `Calling` C_l Cs, where Cs is the list uninitialized classes found so far (with C_l being the most recently added). Once this list is complete, the flag is set to `Called` $C_l\#Cs$ and the class initialization methods are run for the classes in $C_l\#Cs$ in order. If any of these methods throw an uncaught exception a, the flag is set to `Throwing` Cs' a, where Cs' is the list of classes whose methods were not run. If a frame is not in the middle of a class initialization procedure, the flag is `No_ics`.

3 Classes-Touched Collection Functions

In this section we describe the collection functions used by the RTS algorithms whose safety we consider in Section 6. All of these algorithms collect the names of classes touched over the course of a test's execution on a program. Then, when the program is changed, tests whose touched classes did not change are deselected. We have used two different general approaches to this kind of collection. The gist of each is as follows:

- *Naive approach*: At each step of execution, collect every class that might affect that step.
- *Smarter approach*: At each step of execution, collect only those classes that might affect that step that – by virtue of the step type – cannot be safely assumed to be collected during some other step. (That is, use knowledge of contextual guarantees to collect classes less frequently.)

An algorithm using a naive collection function is easier to prove safe, as it is safe over each step of execution. Once its safety is proved, it can be used to prove the safety of a correctly-defined smart collection function by showing it collects the same classes. For this reason, we will give instances of both, using proof of safety of the first to prove safety of the second.

3.1 A Naive Algorithm

The naive approach is to at each step collect every class that might change the behavior of the step. If an object is used, then the algorithm collects the name of the object's class, and all its superclasses. For example, if a static method is called, it collects the referenced class, and all its superclasses. If done sufficiently thoroughly, it is easy to see why this approach collects enough classes to form the basis of a safe deselection algorithm. Since for each step the classes that could have changed that step haven't changed, the behavior of each step is the same.

Below we give the details of the algorithm. When a class is collected, its superclasses are also collected. The collection goes as follows:

(i) At each step of execution, collect the error classes and the current class of each frame in the frame stack.

(ii) Additionally, collect based on the initialization call status of the current frame:
  (a) `Calling` C Cs: collect C
  (b) `Called` $(C\#Cs)$: collect C and the class defining C's `clinit` method
  (c) `Throwing` Cs a: collect the class of the object found at address a on the heap
  (d) `No_ics`: collect based on the current instruction:
  - `New` C, `Getstatic` C F D, `Putstatic` C F D, or `Invokestatic` C M n[7]: collect C
  - `Getfield` F C, `Putfield` F C, `Checkcast` C, `Invoke` M n, or `Throw`[8]: collect the class of the calling or thrown object, if properly provided;[9] otherwise, collect nothing
  - For any other instruction, collect nothing

Note that this algorithm does not depend directly on the behavior of the JVM. While its behavior runs in parallel to the semantics of the JVM described in Section 2.2.2, the collection at each step is only dependent on that step's initial state. This will be important in Section 4, which describes how this semantics-independent algorithm can be combined smoothly with a small step semantics in order to give an instrumented semantics.

However, while this approach collects the minimal set of touched classes, it also does so by collecting the same classes over and over again unnecessarily. This insight leads to a smarter approach.

3.2 A Smarter Algorithm

The smarter approach recognizes a few things about correct programs:

(i) If an object is used, then that means that the object was created by a `new` instruction.

(ii) Instructions touching classes directly (`new` and the static instructions) will necessarily result in that class (and its superclasses) being initialized before the instruction is resolved.

(iii) The current class of any frame will be either the initial class or the defining class of that frame's current method; in the former case the class was initialized at the beginning of the current test's execution;[10] in the latter case it will have been touched during the method call that created the frame.

(iv) If the initialization call status of a frame is `Called` $(C\#Cs)$, then at some point it was `Calling` C Cs.

From these observations it follows that many uses of a class by instructions can actually guarantee that the class has been used previously, and as such has already been collected. By not collecting at any point that can make this guarantee, the number of places where a class must actually be collected is significantly reduced. The result will be collecting the same set of classes, but each will be collected many fewer times, meaning less added overhead. The modified collection approach is as follows:

- Collect a class when the class initialization procedure is called on it (that is, a frame has initialization call status flag `Calling` C Cs for some Cs)
- On a `getstatic`, `putstatic`, or `invokestatic` instruction:
  · If the field/method does not exist, collect the referenced class and all its superclasses
  · If the field/method does exist, collect all classes between the referenced class and the declaring class (including the referenced class but not the declaring class)
- Collect the names of error classes and their superclasses

Note that when the class initialization procedure is called on a class C, C is collected at the beginning of the procedure, not C's class initialization method. (Recall the difference between the class initialization procedure and a class's initialization method from Section 2.2.1: the former is the steps followed by the JVM that leads to running a class's initialization method and that of its superclasses, whereas the latter is an actual method of a class.) This is necessary because during the procedure, C's superclass is initialized first (if it has not been

[7] `New`'s argument is as described in Section 2.2. `Getstatic`'s and `Putstatic`'s arguments are the calling class, C, the field being referenced, F, and the class defining the field, D. `Invokestatic`'s arguments are the calling class, C, the method being invoked, M, and the number of arguments to the method, n.

[8] `Getfield`'s and `Putfield`'s arguments are as described for `Getstatic` in Section 2.2. `Checkcast`'s argument is the class being cast to, C. `Invoke`'s arguments are the method being invoked, M, and the number of arguments to the method, n.

[9] In the JVM, these instructions all expect an address to an appropriate object to exist in a designated location on the stack. If the expected object is not present, an error is thrown. Note that if the object is missing, then no classes are actually touched or affect the outcome of the instruction besides the error's class.

[10] In Java, to run a program (or test) is actually to run a static method of a class. This class is the initial class, which is initialized before its static method is run.

already); if the superclass's initialization method fails, C's is never run. However, even in that case C must be collected because a change to C could include changing the name of its superclass.

This algorithm is constructed by not collecting anywhere a class is guaranteed to have been collected by a previous use. Furthermore, each of these previous uses is either still a collection point or is covered by its own previous use that is still a collection point. Thus it can be seen that the above collects the same classes as the naive algorithm does. Therefore, as long as the naive algorithm is safe, this smarter algorithm is as well. We will formally prove these observations in Section 6. However, some informal reasoning follows, touching on each place where the naive algorithm collected classes.

First, the collection of error classes only happens once in this approach instead of during every step. The naive algorithm only needed to collect these classes at every step because it was designed in a way that made every individual step clearly safe by itself. For this algorithm, proving safety necessarily involves confirming that classes were collected at some point during execution, so collecting once at the start is sufficient. In practice, these classes would not necessarily need to be collected even then, as they would be initialized. Collecting these classes up-front is only necessary here as an artifact of the way that Jinja handles the error classes (by instantiating them up front).

Second, as noted in the above list of observations, each frame's current class is the class that declared the method whose execution is being handled by that frame. Other than the initial frame, each frame is created by an `Invoke` or `Invokestatic` instruction. In either case, the class declaring the invoked method is collected by other cases. The initial frame's class is the initial class; this class is initialized before the initial method is executed, and so is collected at that point.

Third, the `Called` and `Throwing` initialization call statuses do not need to collect anything because the former is covered by collection at `Calling` (which is guaranteed to have been the init call status in a previous step of execution), and the latter is covered by its use of an existing object (created by a `new` instruction).

Finally, when a static instruction runs, the existence of the field or method being used is checked before the initialization status of the declaring class. Thus if the field or method cannot be found ("does not exist"), all the classes that might affect that existence must be collected, since no class initialization procedure will be called. If the field or method is found, then the initialization procedure is called on the class that declares it. The declaring class will be collected at the beginning of the procedure as per the previous collection rule, as will all its superclasses (either by being initialized during this procedure call, or during whatever procedure call initialized them previously). Thus, of the referenced class and its superclasses, only the classes between the referenced and declaring classes will not be collected via the procedure, and so must be collected at this point. `New` C always results in the initialization of C and its superclasses, so it does not need to collect anything.

No other instructions need to collect any classes at all. This follows from the observations given at the beginning of this section, as all the other collecting instructions use an object created by a `new` instruction (which in turn is preceded by initialization of the relevant classes).

To sum up, if the name of a class is collected at the beginning of its class initialization procedure, then the only other times it needs to be collected are when it is able to change behavior but is not initialized or checked for initialization status. The only places where this can occur are those where a class is referred to in order to access a field or method declared by one of its superclasses.

3.2.1 Ekstazi's Algorithm

In Ekstazi's algorithm, as in the smarter algorithm, only certain uses trigger collection, relying on the fact that some uses are necessarily preceded by another use of the class. However, unlike in the algorithms described above, which are performed via instrumentation of the semantics of the JVM, Ekstazi's algorithm adds print statements to the code just before class uses. The smarter algorithm partially works because the semantics is being instrumented rather than the code, meaning it is possible to instrument the class initialization procedure to collect classes in a way code instrumentation does not allow (as calls to the class initialization procedure occur dynamically during runtime rather than through instructions in the code). Thus Ekstazi's collection function requires a few adjustments from ours in order to achieve the same effect. These adjustments amount to replacing collecting at the beginning of initialization with collecting just before any place where the initialization procedure might be run.

Even assuming instrumentation of the semantics rather than the code, in Ekstazi multiple tests may be run on the same JVM, meaning that some classes may already be initialized at the beginning of any particular test. In such cases, it would be necessary to collect in all places where initialization could be called regardless, as the actual calls would only occur on classes not already initialized. Running multiple tests in the same JVM does introduce the problem of state pollution: the values of static fields may be changed by one test and used by another, potentially affecting the latter's result. We do not consider the effect of state pollution in this paper.

Also worth noting are the two places Ekstazi collects that the Jinja approach cannot due to incompleteness of semantics: reflection invocations and interface invocation. (Neither are modeled in Jinja or its extensions.) According to the JVM specification, the class initialization procedure is called upon "invocation of certain

```
//test class
class T{
  public static void main(...){
    //static method call; calls initialization procedure on declaring class C
    C.M();
  }
}

class D{
  static { throw e; } //class initialization method
}

class D'{
  //class initialization method is empty by default
}

//ORIGINAL DEFINITION FOR CLASS C
 //when initialization procedure is called on C, init procedure is called on D prior to
  running C's init method; D's init method throws an error, so C's init method never runs
class C extends D{
  static void M(){ }
}

//CHANGED DEFINITION FOR CLASS C
 //direct superclass is updated; init procedure will now be called on D' instead of D, so
  test completes without error
class C extends D'{
  static void M(){ }
}
```

Fig. 1. Example of code where Ekstazi does not collect enough

reflective methods." Thus it is unnecessary to collect at these invocations if collection occurs at the beginning of the class initialization procedure (as in the smarter algorithm), and necessary to collect there otherwise (as in Ekstazi). As for interfaces, the `invokeinterface` instruction does not result in calling the initialization procedure, so it is necessary to instrument this instruction in a context with interfaces regardless of semantic or code instrumentation. We leave this addition to future work. [11]

In summary, this is what Ekstazi's algorithm should do:

(i) Collect the classes touched by the following instructions via print statements added prior to them:
  - `new`, `getstatic`, `putstatic`, `invokestatic`, `invokeinterface` instructions
  - Invocations of reflective methods

(ii) Collect all error classes (or rather, treat them all as touched classes for all tests)

In Section 6.3, we will demonstrate the safety of this approach.

Ekstazi's actual collection function (as described in [3]) differs mainly in that, instead of collecting before `new` and `invokestatic` instructions or the initialization procedure, it collects at the beginning of class initialization methods, constructors, and static methods. These collections are too late, making it not safe. For example, consider the code given in Figure 1. When the test is run with the original code for C, Ekstazi will collect class D, as its initialization method is run, but not C, as the error thrown by D's initialization method ends the program before C's initialization method or static method M are run. Our smart algorithm, on the other hand, will collect C when its initialization procedure is called. The test fails under the original code and succeeds under the changed code, so it should be run again. However, the only change is to class C, meaning Ekstazi would not rerun the test.

[11] As interfaces act a great deal like classes that cannot be instantiated, including that they are initialized in the way classes are, we do not believe the addition of interfaces would create any problems with the proofs presented here if handled in the same way. That is, in the smarter algorithm's collection, an interface's name would be collected when the initialization procedure is called on it. In Ekstazi's algorithm, it would be collected before the invocation of interface methods and when a class extending it is collected.

4 Collection Semantics

The addition of information collection like that of the class-collecting algorithms described in Section 3 can be modeled by instrumenting a semantics of the relevant language by adding the collection of relevant information on top of the step's normal behavior.

One approach to this is to directly modify an existing semantics of the relevant language, adding an extra piece to the state that keeps track of the information being collected. However, this approach results in a semantics with no immediate proof of mathematical equivalence in behavior to the original semantics. Proof of this equivalence is required in order to use its consequences. As consequences include a guarantee that results proven over the instrumented semantics hold over the original, proven equivalence is essential to the usefulness of the new semantics. Further, the instrumented semantics would need to be kept equivalent manually.

A better approach is to create a function that takes a semantics and a collection function and produces an instrumented semantics, which we will refer to as a Collection Semantics. This allows a general theorem about the function showing behavioral equivalence between the original and instrumented semantics. Then on any input, this equivalence would be immediate, and any results proved on the former would be instantly applicable to the latter. Also, any changes to the original semantics would be reflected in the instrumented semantics without any extra effort.

Such a function is best defined in Isabelle by using a locale, a way to define a collection of components with a set of axioms on those components. This definition can then be instantiated, giving instances access to any theory developed from the axioms. Further details about locales are given in Section 4.3.

Once instantiated with the naive and smart algorithms given in Section 3, the behavior of the semantics produced can be evaluated and compared, allowing us to prove their safety as a mechanism of test deselection.

We define the `CollectionSemantics` locale by first defining the `Semantics` locale as a base.

4.1 `Semantics` Locale

We first give a general definition for a semantics.

Definition 4.1 A ***Semantics*** is a pair:

- a small-step function `small` that takes a program and a state and returns a set of next states, and
- a set of end states, `endset`,

which fulfills the axiom `endset_final`: $\forall \sigma \in$ `endset`. $\forall P.$ `small` $P\ \sigma = \{\}$.

Note that this definition specifies small-step style semantics. This allows the collection function to be more semantics-agnostic: it will only need to collect based on a state assuming a single step.

Given a `Semantics`, a big-step semantics function `big` is derived from `small` using `endset`: `big` just applies `small` to the input until a state in `endset` is reached, then returns that end state.

4.1.1 Running Example: `Semantics` Instance

The semantics locale can be instantiated with the JVM `exec` function as `small`, with `endset` as the set of states that have empty frame stacks or an exception flag. Recall from Section 2.2.2 that `exec` returns no next state on any of the states in this set, satisfying the `Semantics` axiom `endset_final`.

4.2 `CollectionSemantics` Locale

Given the `Semantics` definition, it is then possible to extend to a `CollectionSemantics`.

Definition 4.2 A ***CollectionSemantics*** is a Semantics paired with a three-tuple:

- a collection function `collect` that takes a program and two states (before and after), and returns a collection,
- a function `combine` for combining collections that takes two collections and returns another, and
- an identity for the combining function, `collect_id`,

where `combine` is associative and `collect_id` acts as both a left- and right-identity under `combine`.

In the above, a “collection” can by anything from a set to an integer to a file.

The pieces `small` and `collect` of a `CollectionSemantics` are used to define a small-step instrumented semantics `csmall`, then extended with `endset` to an instrumented big-step semantics `cbig`. The former simply returns a set of pairs of results returned by applying `small` to the input, then applying `collect` to the input and output. The latter returns the result of applying `csmall` to the input as many times as it takes to reach an end state, using `combine` to combine the information collected across the steps. Note that the resulting collection is the identity `collect_id` if no steps are taken. As the states returned by `csmall` are the same as those returned by `small`, the states returned by `cbig` are also the same as those returned by `big`. Then any proven instance of the definition will immediately be able to use both the derived `cbig` and the result that its output is the same as the derived `big`.

4.2.1 Running Example: `CollectionSemantics` Instances

The instance of `Semantics` given in Section 4.1.1 can be extended to instances of `CollectionSemantics` with the naive and smart class collection functions described in Section 3. Since these functions return sets of classes, the components `combine` and `collect_id` are the set union operator and the empty set, respectively. It is easy to see that the axioms of associativity and left- and right-identity hold.

4.3 Using Locales

In Isabelle, definitions comprised of a collection of fixed items ("components") together with a set of axioms can be given using a `locale`. Once these components and axioms are given, theory can be developed that relies on them, including definitions and lemmas. One might write a definition depending on the components and then prove things about that definition given the axioms. This approach was used to turn the above definitions of `Semantics` and `CollectionSemantics` into a locale, as well as those definitions given in Section 5.

Locales can be instantiated by giving concrete definitions of the correct types to match its components, followed by a proof that this instance of the components meets the requirements mandated by the axioms. Once this has been done, the locale's theory can be used, including any derived definitions it contains and any theory developed about them, in addition to any lemmas proved to follow from the axioms.

5 Formally Defining Regression Test Selection

We will now give a formal, general definition for RTS algorithms. We will then combine this definition with `CollectionSemantics` from Section 4 to get a definition for a collection-based RTS algorithm.

5.1 `RTS_safe` Locale

The following defines a general regression test selection algorithm that is safe.

Definition 5.1 An ***RTS_safe*** is a five-tuple:

- set of valid programs `progs`,
- set of valid tests `tests`,
- output function `out` that takes a program and test and returns a set of program outputs,
- equivalence relation `equiv_out` over pairs of program outputs, and
- deselection relation `deselect` taking an initial program, program output, and altered program,

which fulfills the following axioms:

- `existence_safe`: for all $P, P', t, o1$, if $P, P' \in$ `progs`, $t \in$ `tests`, $o1 \in$ `out` $P\ t$, and `deselect` $P\ o1\ P'$, then $\exists o2 \in$ `out` $P'\ t$. `equiv_out` $o1\ o2$,
- `equiv_out_equiv`: `equiv_out` is an equivalence relation, and
- `equiv_out_deselect`: if `equiv_out` $o1\ o2$ and `deselect` $P\ o1\ P'$, then `deselect` $P\ o2\ P'$.

The sets of valid programs and tests give a scope to the safety axiom: the algorithm is only required to be safe over these given sets.[12] The output function provides some sort of output given a program and test, such as the output of a semantics for the language, as used to run tests with programs. The equivalence relation over outputs gives a way to directly compare outputs to determine whether they count as sufficiently similar results in the context of safety. The deselection relation is the meat of the algorithm, choosing which tests not to run based on a pair of programs, plus an output. As given in the safety axiom, these would be instantiated with the original program, the new program, and the output of running a test over the original program. Then deselection would be applied to the test that produced the given output. This function takes a test output instead of a test because deselection will be based on the achieved output, as there may be more than one.

The safety property we use here is one we call *existence safety.* This version of safety is designed with non-deterministic semantics in mind: if a test may produce more than one outcome, it only guarantees that if the original output of a test resulted in its deselection, then there is at least one equivalent outcome under the

[12] Note that this is desirable rather than scope-reducing because the only programs and tests that are relevant are those that can be run – those that meet well-formedness conditions that would generally be enforced by compilers.

changed program. Under this definition, if a flaky test (i.e., one that can produce both a passing and failing outcome under the same program) is deselected, this axiom guarantees it will remain flaky under the changed program. We have chosen this definition of safety because the algorithms we describe here are not designed to identify flaky tests to rerun. Thus, this is the kind of safety promise that is expected and desired here.

The two axioms other than safety require that `equiv_out` is in fact an equivalence relation over outputs and is fine-grained enough that equivalent outputs are indistinguishable to the `deselect` function. When combined, these axioms are sufficient to prove the following:

Lemma 5.2 Safety Transitivity: *If a non-empty sequence of programs Ps are all in* `progs`*, test t is in* `tests`*, o_0 is an output under the first program in Ps and t, and o_0 is deselected under each sequential pair of programs in the sequence, then there is an output under the final program in Ps and t that is equivalent to o_0.*

This lemma is a guarantee that after a deselection based on a given output, it is safe to continue using that output for future deselection decisions until the test is selected again. This is important because the intention of an RTS algorithm is to not run a deselected test again until it is selected, meaning that there will be no updated output to use until this occurs.

As described in Section 4.3, `RTS_safe` can be turned into a locale.

5.2 *`CollectionBasedRTS`* Locale

Having defined `CollectionSemantics` (Section 4.2) and `RTS_safe` (Section 5.1), we are able to define the combination of the two, `CollectionBasedRTS`. This gives the general form of an RTS algorithm that uses information collected during execution to make selection decisions.

Definition 5.3 A ***CollectionBasedRTS*** is a triple of a CollectionSemantics; an RTS_safe whose `out` returns a set of state-collection pairs; and the pair:

- a function `make_test_prog` that takes a program and a test and returns a modified program that includes the ability to run the test, and
- a function `collect_start` that takes a program and returns a starting collection,

which fulfills the axiom `out_cbig`: $\forall P\, t.\ \exists \sigma.\ \texttt{out}\ P\ t = \{(\sigma', coll') \mid \exists coll.\ (\sigma', coll) \in \texttt{cbig}\ (\texttt{make\_test\_prog}\ P\ t)\ \sigma\ \wedge\ coll' = \texttt{combine}\ coll\ (\texttt{collect\_start}\ P)\}$.

While `CollectionSemantics`'s `cbig` takes a program and a state, `RTS_safe`'s output function takes a program and a test as inputs. The former is a general execution function allowed to start at any point in execution. The latter just runs tests, deriving a start state for execution from the given test and program. Therefore the latter can be seen as a specific instance of the former. `CollectionBasedRTS`'s components and axioms are defined around formalizing this idea.

The function `make_test_prog` takes a program and a test as might be given to `out` and returns a program for input into `cbig`. The function `collect_start` returns a collection for each program representing the information that should be collected about it up-front. This represents any information that the RTS algorithm takes into account on the basis of the program itself, and which the `out` function will include automatically.

The axiom formalizes the above expectations of the relationships between `out`, `cbig`, `make_start_prog`, and `collect_start`. For every program-test pairing, there exists a state σ such that the outputs of `out` and `cbig` are equal if the program's starting collection is added to the latter's collection outputs. The state σ is functionally the state that the `out` function runs from on the given program and test.

5.2.1 *Running Example: `CollectionBasedRTS` Instances*

The instances of `CollectionSemantics` given in Section 4.2.1 can be extended to instances of `CollectionBasedRTS` with the following additional instanstiations:

`make_test_prog` is a function that takes a program (a list of class definitions) and a test (a class definition) and adds the test class to the beginning of the program's list to make a new program.[13]

`collect_start` always returns the empty set in the naive case; since each step collects everything for that step, nothing is collected up-front. In the smart case, `collect_start` collects the exception classes and their superclasses.

[13] It also creates a class definition for a `Start` class whose superclass is `Object` and has two methods: a class initialization method that does nothing, and a `main` method that calls the test class's `main` method. This class simplifies modeling the calling of the class initialization procedure on the test class, which is the true initial class, by creating what is essentially a "nothing" frame from which the procedure can be called and to which it can return for the completion of the call to the test's `main`.

`out` takes a program and a test, and applies `cbig` (derived from the `CollectionSemantcs` being extended) to the program returned by `make_test_prog` on the given program and test and the starting state dictated by the same.[14] By design, this function meets `CollectionBasedRTS`'s required relationship with `cbig`.

`deselect` takes a JVM program; a (JVM state)-(class collection) pair; and a second program, and returns `True` if the classes in the collection have not changed from the first program to the second,[15] `False` otherwise.

`equiv_out` is defined as equality between (JVM state)-(class collection) pairs. This definition clearly meets `RTS_safe`'s axioms as an equivalence relation where equivalence outputs are indistinguishable by `deselect`.

`progs` is the set of JVM programs that are well-formed, do not already contain a `Test` or `Start` class,[16] and whose `Object` class's `main` method - if it exists - is static, takes no arguments, and returns type `void`.

`tests` is the set of class definitions whose name is `Test`, create well-formed programs when combined with any of the programs in the above described set, and have a `main` method that is static, takes no arguments, and returns type `void`.

For both instantiations, `RTS_safe`'s existence safety is the only axiom that is not immediate. Proofs of this axiom for both the naive and the smart instance will be presented in Section 6.

6 RTS Safety Proofs

Below we will describe the steps taken in formally proving safety of the naive and small collection-based RTS algorithms. All lemmas and theorems stated in this section are in the context of JinjaDCI's JVM semantics.

6.1 Safety of Naive Algorithm

Proving the safety of the naive collection-based deselection algorithm boils down to showing that for each kind of step of execution, the classes collected by that step are the only classes that could affect its behavior. In other words, if those classes are unchanged in a changed program, the behavior of the step remains the same.

Lemma 6.1 Naive Single-Step Safety*: If the classes collected by the naive collection function over the single step of JVM execution under program P from valid state σ do not differ between programs P and P', then the single step of execution under P' from state σ yields the same state and collection.*

Lemma 6.1 can then be extended from one step to many. From this and the validity of the start state, it is straightforward to show that the end state reached from the start state will be the same under any two programs that agree on the classes collected over the full execution.

Theorem 6.2 Naive Safety*: If P and P' are well-formed JVM programs, t is a valid test class, (σ, Cs) is an output of the naive collection JVM semantics under P and t, and the classes in Cs do not change from P to P', then (σ, Cs) is an output of the naive collection JVM semantics under P' and t. Thus deselecting t on this basis is safe.*

Well-formed JVM programs and valid test classes are as defined for `progs` and `tests` in the naive instantiation of the `CollectionBasedRTS` locale. Note that when the classes in the collection Cs are unchanged from P to P', that is when the naive algorithm will deselect t. Thus the stated safety of t's deselection in this case is also safety of the naive collection as a method for deselection.

6.2 Safety of Smarter Collection

The approach to proving the safety of the smarter collection-based deselection algorithm is necessarily less direct than that for the naive approach. By design, most of the classes that could affect a given step of execution are not collected at that step, relying instead on being collected by either an earlier or later step in execution. For each step of execution, represented by the current state at that step, the classes collected at earlier or later steps can be grouped into "backward promises" - classes collected prior to the step - and "forward promises" - classes that will be collected in future steps, if they have not already been collected. Which classes are in each promised set are determined by the state in that step of execution.

The *backward-promised* classes relative to a state are the initialized classes (as marked on the static heap), classes of objects in the heap, current classes from the frame stack (i.e., those declaring the methods that are currently mid-execution), classes of system exceptions, and superclasses of all the above. The *forward-promised*

[14] Given a program, the start state starts with a starting heap (which has starting instances of the error classes), a starting frame stack (with a single frame whose class and method are `Start` and `main`, with program counter 0 and initialization call status `No_ics`), and the starting static heap (that simply sets `Start`'s initialization state flag to `Done`).

[15] A class is considered changed is anything inside it is different, or if it exists in one program and not the other.

[16] This requirement is an artifact of Jinja's JVM requiring class names to be unique; in practice this is not restricting.

classes depend on the current initialization call status. If it is `Calling` C Cs, then C (i.e. the class whose initialization is actively being called by the current frame) and its superclasses are promised. If it is `No_ics` or `Called` [], then the class whose initialization is checked by the current instruction's execution (if the instruction is a `new` or static instruction) and its superclasses are promised.

The backward-promised classes are designed to cover those classes that are known to have been collected based on information currently present in the state. These classes are most of those we had previously observed could be counted on having been previously collected: classes that have already been initialized, classes that have been instantiated, and so on. These promises, once proved, allow proof that instructions that, for example, use an initialized class or an existing object on the heap, do not have to collect those classes.

The forward-promised classes are those that are about to be collected during an initialization procedure, if they have not been already. This promise, once proved, allows proof that steps in the middle of the initialization procedure do not have to already have collected the superclasses of the class currently being initialized.

Together, the promises are designed so that in order to show that smart collection collects at least the classes collected by naive collection, it is sufficient to show that these promises are kept. First, we show that the forward-promised classes not covered by backward promises are collected:

Lemma 6.3 Forward Promises Kept*: If `Object` is a superclass of C in program P, σ is a non-end state whose top frame has initialization call status `ics`, and:*

- *`ics` is `Calling` C Cs, or*
- *`ics` is `No_ics` and the current instruction of the top frame of σ is `New` C, `Getstatic` D F C, `Putstatic` D F C, or `Invokestatic` D M n, where F or M (as applicable) exists, is static, and is seen by D in C*

then all classes of C and its superclasses that are uninitialized on σ's static heap are collected by the smart collection algorithm by the end of complete execution from σ.

Note that Lemma 6.3 only promises collection of classes that are uninitialized on the static heap. This is because all initialized classes are guaranteed collected by the backward promise about classes on the static heap. Additionally, the case where the current initialization call status is `Called` [] is not covered because state conformity[17] guarantees that the class whose initialization was called is initialized on the heap, meaning that the backward promise for classes on the static heap is sufficient for the promise to be kept. The requirements for existence of the relevant static method or field are there because initialization will only occur if these checks pass. (Naive collection also skips these classes if the exists or static checks fail.) Finally, note that the classes promised collected by this lemma can only be assumed collected when execution terminates, as it only guarantees the classes collected by the end of execution.

The two pieces of Lemma 6.3 are proved separately: first the `Calling` case is proved by induction over the steps of execution. The other case is then proved for each relevant instruction type, using the first case and that the next execution step after each will be to set the current initialization call status to `Calling` C [].

Unlike the forward promises, the backward promises are a preservation property. That is, for each step of execution, if the backward promises have been kept up to that point, then that step will preserve those promises. More precisely, since backward promises are entirely relative to the current state, if the backward promises are assumed kept at a state σ via collection Cs, then if execution of that state yields state σ' and some collection Cs', then the backward promises are kept for state σ' by the combined collection $Cs \cup Cs'$. So if, for example, a step of execution adds an object to the heap - increasing the scope of the promise that heap object's classes are collected - the class of that object is either collected by that step (and is in Cs') or is already covered by σ's backwards promises (and is in Cs). Either way, it will be in $Cs \cup Cs'$.

This preservation of the backward promises is formally stated in the first half of Lemma 6.4 below. That fact, together with Lemma 6.3's guarantees about forward promises being kept, can be used to prove that the smart collection algorithm collects at least those classes collected by the naive collection algorithm (the second half of Lemma 6.4):

Lemma 6.4 *If P is a well-formed JVM program under the typing Φ,[18] the state σ is fully conforming under P and Φ, and the set of classes Cs contains all classes described by the backward and forward promises over σ plus the classes collected by the smart algorithm on the single execution step under P from σ to σ', then:*

(i) Backward Promise Preservation*: The classes described by the backward promises over σ' are in Cs*

[17] Our proof of equivalence between smart and naive collection, and thus safety of smart collection, assumes a well-formed program. We have further shown that well-formed programs produce conforming states (and execution preserves state conformance, so it can be safely assumed here). For a complete definition of state conformance, `correct_state`, see JinjaDCI's `BVConform` theory.

[18] Φ is a function that returns the expected types for the stack and local variables for each instruction of each method of each class in a program. It is used by Jinja and its extensions to encode expected types in a way that allows proof of type safety of their JVM byte code semantics.

(ii) Naive $\subseteq$ Smart*: Since backward and forward promises are kept, all the classes collected by the naive algorithm are in Cs*

Further, by definition inspection, the naive algorithm collects at least all the classes collected by the smart algorithm (Smart $\subseteq$ Naive). Thus, together with Lemma 6.4 ii, the smart and naive algorithms collect the exact same set of classes during the execution-to-termination of a well-formed JVM program starting from a state whose backward promises are met by the starting collection set. (The start state's backward promises must be met up-front so that backward promise preservation can kick in.) Since the smart algorithm's starting collection set is designed to meet the backward promises of its start states, we get the following:

Lemma 6.5 Naive = Smart*: If P is a well-formed JVM program and t is a valid test class, then the set of classes collected by running the naive-instrumented JVM semantics over P and t is equal to the set of classes collected by running the smart-instrumented JVM semantics over P and t.*

Therefore, since the naive approach is safe (Theorem 6.2), the smart approach is as well.

Theorem 6.6 Smart Algorithm Safe*: If P and P' are well-formed JVM programs, t is a valid test class, (σ, Cs) is an output of the smart collection JVM semantics under P and t, and the classes in Cs do not change from P to P', then (σ, Cs) is an output of the naive collection JVM semantics under P' and t. Thus deselecting t on this basis is safe.*

6.3 Making Ekstazi Safe

In Section 3.2.1, we described the differences between the smart collection algorithm and that used by Ekstazi and presented a modified set of collection points for the latter. In order to achieve the safety guaranteed by the above proofs, Ekstazi must at least collect in places that cover what we have outlined here. In particular, since Ekstazi cannot collect directly during the class initialization procedure, it instead collects at each instance where the procedure will be called - in advance of the call. After the call would be too late, as class initialization does not return to the calling instruction if it fails. Since this is what our modified set does, an algorithm using this modified collection function is safe.

6.4 A Note About Threads

The proofs of safety of the algorithms described here are over a semantics that does not include threads. However, class initialization is key to class collection in the smart algorithm and the class initialization procedure uses locks to ensure that classes are not used until they are fully initialized. Thus even though a semantics including threads would be more complicated, if the locking mechanism correctly prevents class use prior to initialization, proof that the collection algorithms given would be safe would follow in a very similar fashion.

7 Related Work

Over the years there have been many approaches and algorithm proposed and implemented for the purposes of regression test selection.(A recent survey of techniques is given in [1].) A number of these techniques are regarded as safe, but with only informal arguments. Formal proof is quite often skipped because it requires a formal model of the language in addition to efforts like presented here. For some cases this can be good enough, as a large number of uses eventually uncover most errors in reasoning. Further, while safety can be desirable, the time trade-off of near-safety can be good enough if the full test suite is occasionally run. Other important features of an RTS technique are precision (minimal tests run with unchanged results; safe but imprecise algorithms mean more time running tests) and inclusivity (the percentage of modification-revealing test cases selected; a safe RTS algorithm is 100% inclusive).

The approach used by Ekstazi was proposed by Skoglund and Runeson [11] as a modification to the *class firewall* technique in order to make it safe. (The class firewall technique involves statically determining the relationship between modules or classes in a program and uses these relationships to determine which tests to run, but can miss tests that run code inside of a firewall when the methods used to determine the initial structure are not reliable [10].) However, the proof of that modification's safety is informal. In their paper [3] on Ekstazi, Gligoric et al. present explanations for their chosen instrumentation points, but they do not present formal proof, which is what we have sought to rectify here.

Collection Semantics as presented here can be thought of as a labeled semantics with a built-in interpretation function over the label trace of an execution. Labeled semantics are generally used for collecting information during execution (as we do here) and have seen many uses (those given in [8,9,6,2] are just some examples). Labeled semantics itself is an instance of a labeled transition system, a construct formalized in Isabelle in [6]. Connecting our Collection Semantics locale to this work (such as by proving it to be an instance of the LTS locale) would be straightforward, but the result would not have advanced any of the goals of this paper.

It could, however, prove useful in allowing simulation of one Collection Semantics by another, especially in attempts to prove that the labels could be correctly replaced by instructions (such as the print instructions used by Ekstazi). We leave this connection and proof to future work.

8 Conclusion and Future Work

In this paper we presented a proof of safety for class-collection based RTS algorithms for JVM programs based on that used by the Java testing library Ekstazi. These proofs were given in the theorem prover Isabelle over the partial Java and JVM semantics JinjaDCI. The first of the algorithms collects classes exhaustively everywhere they were used. The second collects classes when the initialization procedure is called on them and when they are between the referenced and defining classes of static fields and methods called via static instructions. Both differ from Ekstazi's by instrumenting the semantics of the JVM rather than the code run in it. As a code instrumentation, Ekstazi's algorithm cannot collect at actual initialization calls (as they occur at runtime), and replaces this part of our second algorithm with collecting at each instruction that may call the initialization procedure. Thus the safety of our modification of Ekstazi's algorithm can be derived from the safety of our algorithm by seeing that it collects the same set of classes in corresponding places, just slightly earlier when necessary. We also pointed out why this modified set is necessary, thus fixing a bug in their algorithm.

The formalization of the two algorithms' instrumentations is given via defining the Collection Semantics locale. This locale allows the combination of a small-step semantics with a collection function, allowing the latter to be somewhat semantics-agnostic, and the derived semantics to have automatic lemmas of behavioral equivalence with the original.

We leave formal proof of the actual code of a modified Ekstazi to future work, along with the formalization of further aspects of Java's semantics. However, we do not anticipate any of these additions to impact the results presented here. The Collection Semantics locale can be further used to give definitions of various labeled semantics such as those uses mentioned in Section 7. The RTS locale we defined is also sufficiently general to be usable to formally define other RTS algorithms in the context of existence safety.

References

[1] Biswas, S., R. Mall, M. Satpathy and S. Sukumaran, *Regression test selection techniques: A survey*, Informatica (Slovenia) **35** (2011), pp. 289–321.
URL http://www.informatica.si/index.php/informatica/article/view/355

[2] Gadducci, F., F. Santini, L. F. Pino and F. D. Valencia, *A labelled semantics for soft concurrent constraint programming*, in: T. Holvoet and M. Viroli, editors, *Coordination Models and Languages* (2015), pp. 133–149.

[3] Gligoric, M., L. Eloussi and D. Marinov, *Practical regression test selection with dynamic file dependencies*, in: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015 (2015), p. 211–222.
URL https://doi.org/10.1145/2771783.2771784

[4] Klein, G. and T. Nipkow, *A machine-checked model for a java-like language, virtual machine, and compiler*, ACM Trans. Program. Lang. Syst. **28** (2006), p. 619–695.
URL https://doi.org/10.1145/1146809.1146811

[5] Lindholm, T., F. Yellin, G. Bracha and A. Buckley, *The Java Virtual Machine Specification: Java SE 8 Edition* (2015).
URL https://docs.oracle.com/javase/specs/jvms/se8/html/index.html

[6] Lochbihler, A., *Jinja with threads*, The Archive of Formal Proofs. http://afp.sf.net/entries/JinjaThreads.shtml (2007).

[7] Mansky, S. and E. L. Gunter, *Dynamic class initialization semantics: A jinja extension*, in: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019 (2019), p. 209–221.
URL https://doi.org/10.1145/3293880.3294104

[8] Mansky, W., Y. Peng, S. Zdancewic and J. Devietti, *Verifying dynamic race detection*, in: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017 (2017), p. 151–163.
URL https://doi.org/10.1145/3018610.3018611

[9] Nagarakatte, S., J. Zhao, M. M. Martin and S. Zdancewic, *Softbound: Highly compatible and complete spatial memory safety for c*, in: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09 (2009), p. 245–258.
URL https://doi.org/10.1145/1542476.1542504

[10] Rothermel, G. and M. J. Harrold, *Analyzing regression test selection techniques*, IEEE Transactions on software engineering **22** (1996), pp. 529–551.

[11] Skoglund, M. and P. Runeson, *Improving class firewall regression test selection by removing the class firewall*, International Journal of Software Engineering and Knowledge Engineering **17** (2007), pp. 359–378.
URL https://doi.org/10.1142/S0218194007003306

Correct Audit Logging in Concurrent Systems

Sepehr Amir-Mohammadian[a,1] Chadi Kari[a,2]

[a] *School of Engineering and Computer Science, University of the Pacific, Stockton, California 95211*

Abstract

Audit logging provides post-facto analysis of runtime behavior for different purposes, including error detection, amelioration of system operations, and the establishment of security in depth. This necessitates some level of assurance on the quality of the generated audit logs, i.e., how well the audit log represents the events transpired during the execution. Information-algebraic techniques have been proposed to formally specify this relation and provide a framework to study correct audit log generation in a provable fashion. However, previous work fall short on how to guarantee this property of audit logging in concurrent environments. In this paper, we study an implementation model in a concurrent environment. We propose an algorithm that instruments a concurrent system according to a formal specification of audit logging requirements, so that any instrumented concurrent system guarantees correct audit log generation. As an application, we consider systems with microservices architecture, where logging an event by a microservice is conditioned on the occurrence of a collection of events that take place in other microservices of the system.

Keywords: Audit logging, Concurrent systems, Programming languages, Security

1 Introduction

Reliable audit logging is essential to provide secure computation through the after-the-fact analysis of the audit log. Audit logging is used along with preventive security mechanisms to enable in-depth enforcement of security. In-depth enforcement of security refers to multiple layers of pre-execution, runtime, and post-execution techniques to ensure the legitimacy of the computation. Examples abound, e.g., a medical records system that enforces preventive measures including user authentication, static/dynamic information flow analysis to prevent leakage or corruption of data [47], and access authorization, e.g., to deny illegitimate access of certain users to certain medical data. Moreover, the system engages in the collection and a posteriori analysis of audit logs for different purposes, including the satisfaction of accountability goals, e.g., established by Health Insurance Portability and Accountability Act (HIPAA) [16,15], reinforcement of access control [56,12], etc.

Using audit logging along with preventive mechanisms has two major applications: 1) Post-execution analysis of audit logs provides a platform to detect security violations based on the logged evidence [60,5]. This class of logging policies rely on the notions of user accountability and deterrence. 2) Audit logging is used to detect existing vulnerabilities in the preventive security mechanisms and ameliorate those mechanisms [7,44].

In both of the aforementioned applications, effectiveness of in-depth security relies on the correctness and efficiency of the generated audit log and its after-the-fact analysis. Audit log correctness and efficiency reflect on some challenges in the generation of audit logs. Correct audit logging must record factual information about the runtime behavior, which may be ensured by the verification of auditing policies and their runtime enforcement mechanisms. Moreover, a correct audit log must include sufficient information according to what the auditing policy specifies. In addition, efficiency of audit logging must be emphasized in in-depth security in order to improve system performance regarding the collection and analysis of audit logs. Efficient audit logging

[1] Email: samirmohammadian@pacific.edu

[2] Email: celkari@pacific.edu

entails to only record necessary information about the computation, rather than naively collecting *all* events in the log. These issues have been challenging in the wild, for instance resulting in failure to safeguard against data breaches, and are considered as one of the top ten most critical security risks by Open Web Application Security Project [50].

To establish a formal foundation for audit logging, a general semantics of audit logs has been defined [5,6] using the theory of *information algebra* [29]. This line of work helps to study whether the mechanism of enforcing an audit logging policy is correct and efficient on the basis of the proposed information-algebraic framework. Both program execution traces and audit logs are interpreted as information elements in this framework. In essence, the relation between the information in execution traces and the information in audit logs is formulated according to the established notion of information containment. An audit log is defined correct if it satisfies this relation. This formulation facilitates the separation of the specification of auditing requirements from programs, which is of great value in practice. This way, rather than manual inlining of audit logging in the code, algorithms can be proposed that automatically instrument the code with audit logging capabilities. The semantic framework enables algorithms for implementing general classes of specifications for auditing and establish conditions that guarantee the enforcement of those specifications by such algorithms.

The aforementioned line of work relies on the proposed semantic framework for audit log generation whose implementation model is constrained to linear process executions. This limitation, in practice, restricts the application of the framework to systems where a single thread of execution is involved in the generation of audit logs. For instance, in the case study of a medical records system [6], audit logging capability is considered as an extension to the web server program, and all preconditions for logging depend on the events that transpire in the same program execution thread. As an example consider breaking the glass event [32]. Breaking the glass is used in critical situations to bypass access control. By breaking the glass, system users increase their authority in the system in order to gain access to certain data, but simultaneously admit to be accountable for their actions. Breaking the glass event is a precondition to log accesses to particular patient information. Instrumentation of medical records web server guarantees correct audit logging as long as such events occur in the execution trace of the single-threaded web server. This eliminates the possibility of distributing authentication and authorization tasks to other concurrent components of the system. Such restriction encourages us to study the semantics of audit logging in concurrent environments that underlies correct instrumentation of multithreaded and multiprocess applications for auditing purposes. Indeed, real-world examples of inadequate logging and monitoring in concurrent and distributed systems, e.g., a recent security incident in a retailer's network of POS systems [59], demonstrates how crucial it is to ensure the correctness of audit logging mechanisms.

The proposed semantic framework needs to provide a mechanism to specify auditing requirements based on concurrent execution traces. Our framework needs to be general enough to encompass different audit log generation and representation approaches as its instances. The generality of information theoretic models have already been shown in this realm [6]. We demonstrate that such a model can be used for concurrent systems. We use the model to interpret audit logs, specify audit logging requirements and define correct enforcement of such requirements in concurrent systems. Similar to the previous work on linear process executions, correctness of log in concurrent environments is conditioned on the specifications of auditing requirements through the comparison between the information contained in the log and the information advertised by those specifications.

We instantiate our general model with a sufficiently expressive language in order to specify and enforce auditing requirements in concurrent environments. Horn clause logic is a proper language for this purpose due to straightforward modeling of execution traces as sets of facts, sufficient expressivity to specify auditing requirements and available logic programming implementations, e.g., [46,55].

A formal language model is used to specify and establish correct enforcement of audit logging policies in concurrent systems according to the developed framework. We use a variant of π-calculus [35] with unlabeled reduction semantics for this purpose. This formalism provides a model for developing tools with correctness guarantees. This model enjoys the following features.

- Concurrency: In order to specify multithreaded programs and multiprocessor systems, the language model supports concurrent process executions with interprocess communication (IPC) through message passing.
- Generality: While different process calculi are potential choices to describe our implementation model, we use a variant of π-calculus due to its sufficiently concise and high level syntax and semantics to describe interactions among processes. This facilitates formulation of a wide range of concurrent systems.
- Timing: We need to be able to specify the ordering of interesting events for the sake of specifying auditing requirements. For example, in break the glass policies access to particular patient information is logged as long as the glass is already broken. In order to implement such specifications, we need to apply a timing mechanism that is shared among all processes of the system. Each step of concurrent execution of processes updates this universal time.
- Named functions: To specify auditing requirements, a fundamental unit of secure operations is required. Functions can be considered as abstractions of these fundamental units in different languages and systems.

Our language model supports named functions, in terms of sub-agents of each agent.

Using the formalism with aforementioned features enables us to model concurrent environments that guarantee the correct generation of audit logs according to the developed semantic framework. In this paper, we propose an instrumentation algorithm that receives a concurrent system as input and modifies the system according to a precise specification of audit logging requirements. We show that this algorithm is correct (based on the semantic framework), and hence the instrumented concurrent system generates correct audit logs. As implied earlier, enforcement of audit logging policies through code instrumentation separates policy from code, provides a foundation to study the effectiveness of enforcement mechanism using formal methods, and can be applied automatically to legacy code to enhance system accountability.

Since our model is based on process calculi, IPC is handled by message-passing. Modeling alternative IPC approaches for correct audit logging, e.g., shared memory and/or files, is considered as potential future work.

Case studies that benefit from the result of this work include deployment of correct logging capabilities in multiprocess and multithreaded client-server and peer-to-peer applications, microservices, etc. While this paper provides a prototype instrumentation algorithm in abstract settings, as a future work, we aim to deploy our existing instrumentation algorithm in Spring Boot [53], a Java microservices framework, that facilitates code instrumentation through aspect-oriented programming [2].

1.1 Paper Outline

The rest of the paper is organized as follows. In Section 1.2, we discuss an illustrative example for audit logging in concurrent systems and in Section 1.3 we discuss the threat model. Section 2 reviews the information-algebraic semantics of audit logging and instantiation of the model with first-order predicate logic. Section 3 discusses the implementation model in detail. In particular, Section 3.1 introduces the syntax and semantics of the source system, a variant of π-calculus. In Section 3.2, we study a class of logging specifications that assert temporal relations among function invocations, potentially in different concurrent components of a system. Section 3.3 studies the syntax and semantics of the target system. In Section 3.4, we propose our instrumentation algorithm, along with the properties of interest that the algorithm satisfies. The proofs of these properties are given in our accompanying Technical Report [3]. In Section 4, related work is discussed. Finally, Section 5 concludes the paper and specifies future work.

1.2 An Example: Microservices-based Medical Records Systems

In this section, an oversimplified example is given that illustrates the application of audit logging in concurrent environments. We will revisit this example later in the paper (through Examples 3.1 and 3.2) to explain sample instantiations of our formal framework.

Many applications have been shifting their architecture from a traditional monolithic structure toward service-oriented architecture (SOA) in order to boost maintainability, continuous deployment and testing, adaptation to new technologies, system security, fault tolerance, etc. One popular deployment approach to SOA is where an application is decomposed to a set of highly collaborative processes, called microservices. A microservice must be minimal, independent, and fine-grained. Minimality constrains a microservice to access and manipulate certain data types within an application, ideally a single database per each service. Microservice instances run independently in their own containers, virtual machines, or hosts. To accomplish its own goals, a microservice communicates with other microservices of the application through message passing, or remote procedure calls (RPCs). Jolie [25] is the programming language for developing applications with microservices architecture. Its formal semantics [21,36] is defined as a process calculus, inspired by π-calculus.

The need to better streamline healthcare services is pushing medical record systems toward microservices [49]. In fact, a new study shows that microservices-based healthcare is anticipated to experience fivefold increase in market value within the next few years [61].

In what follows we describe a simple example of microservices-based medical records system, where audit logging for certain events is necessary, as dictated by the accountability requirements. Figure 1 depicts a medical records system with microservices architecture that includes Authorization and Patient services (among others). Application front-end includes API gateway that multiplexes user requests (from different clients, e.g., web, mobile applications, etc.) to certain microservices. Patient microservice manages the information about patients, e.g., their medical history. Authorization microservice handles different operations to authorize access to system data, including e.g., breaking the glass.

As mentioned earlier, by breaking the glass, the user agrees to comply with accountability regulations. The common solution to follow accountability regulations is to generate trail of audit logs at runtime. One such audit logging requirement may be as follows: "Any attempt by a healthcare provider to read patient medical data must be stored in the log, if that provider has already broken the glass."

This example demonstrates the core ideas that we are pursuing in this paper:

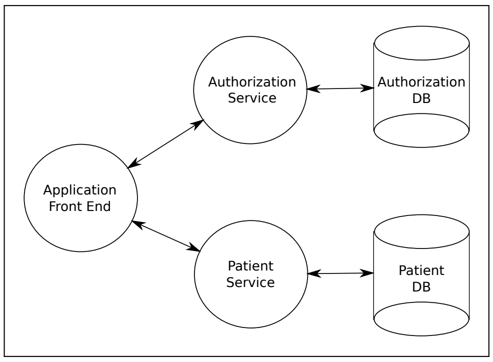


Fig. 1. Authorization and Patient microservices of a medical records system.

- A concurrent system is employed, where each component runs independently and in collaboration with other components, e.g., a medical records system with different microservices including the ones described above.
- Audit logging requirements necessitate logging certain events in a concurrent component, provided that a set of other events have previously occurred in potentially other concurrent components, e.g., logging the event of reading medical history in Patient microservice if the glass is already broken in Authorization microservice.
- We investigate an algorithmic approach to establish correct audit logging for concurrent environments according to the already-established audit logging requirements. We expect correctness of audit logging in our medical records system example, for instance, to imply only logging the reading attempts by the user who has broken the glass. This avoids missing any logging event, as well as logging unnecessary events. We accomplish this by instrumenting Authorization and Patient microservices, and in particular the operations of interest, i.e., breaking the glass and reading patient medical history.

1.3 Threat Model

We assume that the concurrent system subjected to instrumentation is not supporting audit logging in the first place, or is suffering from either insufficient or overzealous audit log generation. However, we assume that the concurrent system that is deployed according to our implementation model, passes both static and dynamic checks, e.g., syntactic checks, type checks, compilation, interpretation, etc. We trust the compiler/interpreter, and the runtime environment in which that system is being executed. Moreover, we trust the implementation of our instrumentation algorithm, and its compilation and/or interpretation, along with the runtime environment in which the instrumentation algorithm is executed. We also trust the integrity of logging specifications that assert audit logging requirements. Finally, we trust the compilation/interpretation process for the instrumented concurrent system that is deployed based on our implementation model, as well as the system's runtime environment. Security of the messages transmitted between concurrent components and the generated audit logs is considered to be out-of-scope.

These assumptions help us to purposefully focus on the essence of logging, i.e., whether logs are generated correctly in the first place and independent of external concerns including reliability of the underlying execution and communication system, latency, etc. which are explored in related work (Section 4).

2 Semantics of Audit Logging

In order to provide a standalone formal presentation, in this section we review the information-algebraic semantics of audit logging and the instantiation of the semantic framework with first-order logic, which is originally proposed by Amir-Mohammadian et al. [6]. We have applied minor modifications to the model to better suit concurrency and nondeterministic runtime behavior, inherent to concurrent systems.

2.1 Information-Algebraic Semantic Framework

In order to specify how audit logs are generated at runtime, we need to abstract system states and their evolution through the computation. A *system configuration* κ abstracts the state of the system at a given point during the execution. Let $\mathcal{K}$ denote the set of system configurations. We posit a binary reduction relation among configurations, i.e., $(\longrightarrow) \subseteq \mathcal{K} \times \mathcal{K}$ which denotes the computational steps, and is used in the standard infix form.[3] A *system trace* τ is a potentially infinite sequence of system configurations, i.e.,

[3] A notational convention throughout the paper is that infix operators and relations are wrapped with parentheses when their signature are specified.

$\tau = \kappa_0\kappa_1\cdots$, where κ_i is the ith configuration in sequence, and $\kappa_i \longrightarrow \kappa_{i+1}$. We denote the set of all traces by $\mathcal{T}$, and define $\mathit{prefix}(\tau)$ as the set of all prefixes of τ.

Information algebra is used to define the notion of correctness for audit logs. In Section 2.2, we instantiate this abstract algebraic structure to model a specific class of audit logging requirements. We define an information algebra in the following.

Definition 2.1 (Information algebra) *An information algebra (Φ, Ψ) is a two-sorted algebra consisting of an Abelian semigroup of information elements, Φ, as well as a lattice of querying domains, Ψ. Two fundamental operators are presumed in this algebra: a combination operator, $(\otimes) : \Phi \times \Phi \to \Phi$, and a focusing operator, $(^{\Rightarrow}) : \Phi \times \Psi \to \Phi$. An Information algebra (Φ, Ψ) satisfies a set of properties, in connection to combination and focusing operators.*[4] *We let $X, Y, Z, \cdots$ to range over elements of Φ, and E range over Ψ.*

$X, Y \in \Phi$ are information elements that can be combined to make a more inclusive information element $X \otimes Y$. $E \in \Psi$ is a querying domain with a certain level of granularity that is used by the focusing operator to extract information from an information element X, denoted by $X^{\Rightarrow E}$. For example, relational algebra is an instance of information algebra, in which relations instantiate information elements, sets of attributes instantiate querying domains, natural join of two relations defines the combination operator, and projection of a relation on a set of attributes defines the focusing operator [29].

Combination of information elements induces a partial order relation $(\preccurlyeq) \subseteq \Phi \times \Phi$ among information elements, defined as follows: $X \preccurlyeq Y$ iff $X \otimes Y = Y$. Intuitively, $X \preccurlyeq Y$ means that Y contains the information element X.

As part of the semantics of audit logging, we treat execution traces as information elements, i.e., the information content of the execution trace. To this end, we posit $\lfloor\cdot\rfloor : \mathcal{T} \to \Phi$ as a mapping in which, intuitively, $\lfloor\tau\rfloor$ refers to the information content of the trace τ. We also impose the condition that $\lfloor\cdot\rfloor$ be injective and monotonically increasing, i.e., if $\tau' \in \mathit{prefix}(\tau)$ then $\lfloor\tau'\rfloor \preccurlyeq \lfloor\tau\rfloor$. This ensure that as the execution trace grows in length, it contains more information.

In the following definition, we define audit logging requirements in an abstract form. We call this abstraction a *logging specification.* This definition is abstract enough to encompass different execution models, as well as different representations of information. In Sections 2.2 and 3.2, we instantiate this definition with a more concrete structure that guides us on how to implement audit logging requirements.

Definition 2.2 (Logging specifications) *Logging specification LS is defined as a mapping from system traces to information elements, i.e., $LS : \mathcal{T} \to \Phi$. Intuitively, $LS(\tau)$ declares what information must be logged, if the system follows the execution trace τ.*

Note that even though $\lfloor\cdot\rfloor$ and LS have the same signature, i.e., maps from traces to information elements, they are conceptually different. $\lfloor\tau\rfloor$ is the whole information contained in τ, whereas $LS(\tau)$ is the information that is supposed to be recorded in the log, if the system follows the execution trace τ.

We denote an audit log with $\mathbb{L}$ which represents a set of data, gathered at runtime. Let $\mathcal{L}$ denote the set of audit logs. In order to judge about the correctness of an audit log, the information content of the audit log needs to be studied in comparison to the information content of the trace that generates that audit log. To this end, we define a mapping that returns the information content of an audit log. We abuse the notation and consider $\lfloor\cdot\rfloor : \mathcal{L} \to \Phi$ as such mapping. Therefore, $\lfloor\mathbb{L}\rfloor$ refers to the information content of the audit log $\mathbb{L}$. We assume that $\lfloor\cdot\rfloor$ on audit logs is injective and monotonically increasing, i.e., if $\mathbb{L} \subseteq \mathbb{L}'$ then $\lfloor\mathbb{L}\rfloor \preccurlyeq \lfloor\mathbb{L}'\rfloor$. Therefore, the more inclusive the audit log is, it contains more information.

The notion of correct audit logging can be defined based on an execution trace and a logging specification. To this end, the information content of the audit log is compared to the information that the logging specification dictates to be recorded in the log, given the execution trace. The following definition captures this relation.

Definition 2.3 (Correctness of audit logs) *Audit log $\mathbb{L}$ is* correct *wrt a logging specification LS and a system trace τ iff both $\lfloor\mathbb{L}\rfloor \preccurlyeq LS(\tau)$ and $LS(\tau) \preccurlyeq \lfloor\mathbb{L}\rfloor$ hold. The former refers to the necessity of the information in the audit log, and the latter refers to the sufficiency of those information.*

A system that generates audit logs at runtime includes the stored logs as part of its configuration. Let the mapping $\mathit{logof} : \mathcal{K} \to \mathcal{L}$ denote the residual log of a given system configuration, i.e., $\mathit{logof}(\kappa)$ is the set of all recorded audit logs in configuration κ. It is natural to assume that the residual log within configurations grows larger as the execution proceeds. The residual log of a trace is then defined using logof.

Definition 2.4 (Residual log of a system trace) *The residual log of a finite system trace τ is $\mathbb{L}$, denoted by $\tau \rightsquigarrow \mathbb{L}$, iff $\tau = \kappa_0\kappa_1\cdots\kappa_n$ and $\mathit{logof}(\kappa_n) = \mathbb{L}$.*

[4] We avoid discussing these properties in detail here for the sake of brevity. Readers are referred to [29] for the complete formulation.

Note that if $\tau \rightsquigarrow \mathbb{L}$, then $\mathbb{L}$ is not necessarily correct wrt a given LS and a trace τ. If the residual log of a trace is correct throughout the execution, then that trace is called *ideally-instrumented*. System trace τ is ideally instrumented for a logging specification LS iff for any trace τ' and audit log $\mathbb{L}$, if $\tau' \in \mathit{prefix}(\tau)$ and $\tau' \rightsquigarrow \mathbb{L}$ then $\mathbb{L}$ is correct wrt τ' and LS. Indeed audit logging is an enforceable security property on a trace of execution [6]. Given a logging specification, ideally-instrumented traces induce a safety property [43], and hence implementable by inlined reference monitors [18], and edit automata [4].

Let $\mathfrak{s}$ be a concurrent system with an operational semantics. $\mathfrak{s} \Downarrow \tau$ iff $\mathfrak{s}$ can produce trace τ', either deterministically or non-deterministically, and $\tau \in \mathit{prefix}(\tau')$. We abuse the notation and use $\kappa \Downarrow \tau$ to denote the same concept for configuration κ. We follow program instrumentation techniques, in order to enforce a logging specification on a system. An *instrumentation algorithm* receives the concurrent system as input along with the logging specification, and instruments the system with audit logging capabilities so that the instrumented system generates the required "appropriate" log. An instrumentation algorithm is a partial function $\mathcal{I} : (\mathfrak{s}, LS) \mapsto \mathfrak{s}'$ that instruments $\mathfrak{s}$ according to LS aiming to generate audit logs appropriate for LS. We call $\mathfrak{s}$ the *source system*, and the instrumented system, i.e., $\mathcal{I}(\mathfrak{s}, LS) = \mathfrak{s}'$, the *target system*. Source and target traces refer to the traces of the source and target systems, resp.

It is natural to expect that the instrumentation algorithm would not modify the semantics of the original system drastically. The target system must behave roughly similar to the source system, except for the operations related to audit logging. We call this attribute of an instrumentation algorithm *semantics preservation*, and define it in the following. This definition is abstract enough to encompass different source and target systems (with different runtime semantics), and instrumentation techniques. The abstraction relies on a binary relation $:\approx$, called *correspondence relation*, that relates the source and target traces. Based on different implementations of the source and target systems, and the instrumentation algorithm, the correspondence relation can be defined accordingly.

Definition 2.5 (Semantics preservation by the instrumentation algorithm) *Instrumentation algorithm $\mathcal{I}$ is semantics preserving iff for all systems $\mathfrak{s}$ and logging specifications LS, where $\mathcal{I}(\mathfrak{s}, LS)$ is defined, the following conditions hold: 1) For any trace τ, if $\mathfrak{s} \Downarrow \tau$, then there exists some trace τ' such that $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'$, and $\tau :\approx \tau'$. 2) For any trace τ, if $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau$, then there exists some trace τ' such that $\mathfrak{s} \Downarrow \tau'$, and $\tau' :\approx \tau$.*

Another related property of the instrumentation algorithm is to ensure that it is *deadlock-free*, meaning that instrumenting a system does not introduce new states being stuck. One approach to define an independent notion of deadlock-freeness is to consider bisimilar source and target traces. Indeed, additional formal constructs must be introduced to translate target traces to source traces for this purpose. Our definition of deadlock-freeness, however, heavily relies on the notion of semantics preservation, and is not a required component in the definition of instrumentation correctness (Definition 2.7). Let source system $\mathfrak{s}$ generate trace τ, and $\mathcal{I}(\mathfrak{s}, LS)$ generate trace τ' such that $\tau :\approx \tau'$. Then, we call $\mathcal{I}(\mathfrak{s}, LS)$ being stuck if $\mathfrak{s}$ can continue execution following τ (at least for one extra step), while $\mathcal{I}(\mathfrak{s}, LS)$ cannot continue execution following τ'.

Definition 2.6 (Deadlock-freeness of the instrumentation algorithm) *Instrumentation algorithm $\mathcal{I}$ is deadlock-free iff for any source system $\mathfrak{s}$, logging specification LS, traces τ and τ', and configuration κ, if $\mathfrak{s} \Downarrow \tau$, $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'$, $\tau :\approx \tau'$, and $\mathfrak{s} \Downarrow \tau\kappa$, then there exists some configuration κ' such that $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'\kappa'$.*

Besides these properties, another important feature of an instrumentation algorithm is the quality of audit logs generated by the instrumented system. The information-algebraic semantic framework provides a platform to define correct instrumentation algorithms for audit logging purposes. Let $\mathfrak{s}$ be a target system, and τ be a source trace. Simulated logs of τ by $\mathfrak{s}$ is the set $\mathit{simlogs}(\mathfrak{s}, \tau)$ defined as $\mathit{simlogs}(\mathfrak{s}, \tau) = \{\mathbb{L} \mid \exists \tau'. \mathfrak{s} \Downarrow \tau' \wedge \tau :\approx \tau' \wedge \tau' \rightsquigarrow \mathbb{L}\}$. Using this set, we can define correctness of instrumentation algorithms in a straightforward manner. Intuitively, the instrumentation algorithm $\mathcal{I}$ is correct if the instrumented system generates audit logs that are correct wrt the logging specification and the source trace. This must hold for any source system, any logging specification, and any possible log generated by the instrumented system.

Definition 2.7 (Correctness of the instrumentation algorithm) *Instrumentation algorithm $\mathcal{I}$ is correct iff for all source systems $\mathfrak{s}$, traces τ, and logging specifications LS, $\mathfrak{s} \Downarrow \tau$ implies that for any $\mathbb{L} \in \mathit{simlogs}(\mathcal{I}(\mathfrak{s}, LS), \tau)$, $\mathbb{L}$ is correct wrt LS and τ.*

2.2 *Instantiation of Logging Specification*

In Definition 2.2, logging specification is defined abstractly as a mapping from system traces to information elements. For a more concrete setting, this definition needs to be instantiated with appropriate structures in a way that is useful in the deployment of audit logging. In essence, we need to instantiate information algebra (Definition 2.1). We are interested in logical specification of audit logging requirements due to its easiness of use, expressivity power, well-understood semantics, and off-the-self logic programming engines for subsets of first-order logic (FOL), e.g., Horn clause logic. To this end, in this section, we instantiate information algebra with FOL, which is expressive enough to specify computational events, and the temporal relation among them.

Indeed, other variants of logic may also be considered for this purpose.

In order to instantiate information algebra, it is required to specify the contents of the set of information elements Φ and the lattice of querying domains Ψ, along with the definitions of combination and focusing operators. Definitions 2.8, 2.9, and 2.10 accomplish these instantiations.

Definition 2.8 instantiates an FOL-based set of information elements. An information element in our instantiation is a closed set of FOL formulas, under a proof-theoretic deductive system.

Definition 2.8 (Set of closed sets of FOL formulas) *Let φ range over FOL formulas, and Γ range over sets of FOL formulas. $\Gamma \vdash \varphi$ denotes a judgment derived by a sound and complete natural deduction proof theory of FOL. We define closure operation Closure as $\mathit{Closure}(\Gamma) = \{\varphi \mid \Gamma \vdash \varphi\}$. Then, the set of closed set of FOL formulas is defined as $\Phi_{FOL} = \{\Gamma \mid \Gamma = \mathit{Closure}(\Gamma)\}$.*

Definition 2.9 instantiates the lattice of querying domains for the FOL-based information algebra. A query domain is a subset of FOL, defined over certain predicate symbols.

Definition 2.9 (Lattice of FOL sublanguages) *Let Preds be the set of all assumed predicate symbols along with their arities. If $S \subseteq \mathit{Preds}$, then we denote the sublanguage $FOL(S)$ as the set of well-formed FOL formulas over predicate symbols in S. The set of all such sublanguages $\Psi_{FOL} = \{FOL(S) \mid S \subseteq \mathit{Preds}\}$ is a lattice induced by set containment relation.*

Lastly, Definition 2.10 instantiates the combination and focusing operators for the FOL-based information algebra. Combination is the closure of the union of two sets of formulas. Focusing is the closure of the intersection of an information element and a query domain.

Definition 2.10 (Combination and focusing in (Φ_{FOL}, Ψ_{FOL})) *Let $(\otimes) : \Phi_{FOL} \times \Phi_{FOL} \to \Phi_{FOL}$ be defined as $\Gamma \otimes \Gamma' = \mathit{Closure}(\Gamma \cup \Gamma')$, and $(^{\Rightarrow}) : \Phi_{FOL} \times \Psi_{FOL} \to \Phi_{FOL}$ be defined as $\Gamma^{\Rightarrow FOL(S)} = \mathit{Closure}(\Gamma \cap FOL(S))$.*

(Φ_{FOL}, Ψ_{FOL}) is an information algebra, given the Definitions 2.8, 2.9, and 2.10.[5] In order to use (Φ_{FOL}, Ψ_{FOL}) as a framework for audit logging, we also need to instantiate the mapping $\lfloor\cdot\rfloor$, introduced in Section 2.1, to interpret both execution traces and audit logs as information elements.

Definition 2.11 (Mapping traces and audit logs to information elements in (Φ_{FOL}, Ψ_{FOL})) *Let $\mathit{toFOL}(\cdot) : (\mathcal{T} \cup \mathcal{L}) \to FOL(\mathit{Preds})$ be an injective and monotonically increasing function. Then, we instantiate $\lfloor\cdot\rfloor = \mathit{Closure}(\mathit{toFOL}(\cdot))$ in order to interpret both traces and logs as information elements in (Φ_{FOL}, Ψ_{FOL}).*

Now we can instantiate logging specification LS in the information algebra (Φ_{FOL}, Ψ_{FOL}). To this end, a set of audit logging rules and definitions are assumed to be given in FOL. Let Γ be this set. Moreover, a set of predicate symbols are assumed that reflect on the predicates whose derivation need to be logged at runtime. This set is denoted by S. A logging specification in this setting, receives a trace τ, combines the information content of τ with closure of Γ, and then focuses on the predicates specified in S. Intuitively, given Γ and S, a logging specification maps a trace τ to the set of all predicates whose symbols are in S, and are derivable given rules in Γ and the events in τ.

Definition 2.12 (Logging Specification in (Φ_{FOL}, Ψ_{FOL})) *Given a set of FOL formulas Γ and a subset of predicate symbols $S \subseteq \mathit{Preds}$, a logging specification $\mathit{spec}(\Gamma, S) : \mathcal{T} \to \Phi_{FOL}$ is defined as $\mathit{spec}(\Gamma, S) = \tau \mapsto (\lfloor\tau\rfloor \otimes \mathit{Closure}(\Gamma))^{\Rightarrow FOL(S)}$.*

3 Implementation Model on Concurrent Systems

In this section, we propose an implementation model for correct audit logging in concurrent systems. To this end, we use a variant of π-calculus to specify the concurrent system, and propose an instrumentation algorithm that retrofits the system according to a given logging specification. We then specify and prove the properties of interest, including the correctness of the instrumentation algorithm (Definition 2.7).

In Section 3.1, the syntax and semantics of the source system model is introduced. Section 3.2 proposes a class of logging specifications that can specify temporal relations among computational events in concurrent systems. Section 3.3 describes the syntax and semantics of the systems enhanced with audit logging capabilities. Lastly, in Section 3.4, we discuss the instrumentation algorithm and the properties it satisfies.

3.1 Source System Model

We consider a core π-calculus as our source concurrent system model, denoted by Π. One major distinguishing feature of π-calculus is modeling mobile processes using the same category of names for both links and

[5] The reader is referred to [4] for the detailed proof, which relies on the properties of natural deduction system of FOL.

transferable objects, along with scope extrusion. However, mobility is not used in our implementation model. Therefore other seminal process calculi e.g., CSP [23] and CCS [34] can also be considered for this purpose. We employ π-calculus due to its concise syntax and simple semantics that provides a clean and sufficiently abstract specification of the required interactions among concurrent components of the system. The syntax and semantics of the source system are defined in the following. It is based on the representation of the calculus given in [38] which deviates from standard π-calculus by dropping silent prefixes, unguarded summations and labeled reduction system, for the sake of simplicity and conciseness.

3.1.1 Syntax

Let $\mathcal{N}$ be the infinite denumerable set of names, and $a, b, c, \cdots$ and $x, y, z, \cdots$ range over them.

Prefixes Prefixes α are defined as $\alpha ::= a(x) \mid \bar{a}x$. Prefix $a(x)$ is the input prefix, used to receive some name with placeholder x on link a. Prefix $\bar{a}x$ is the output prefix, used to output name x on link a.

Agent names and processes Let A, B, C, D range over agent names, and $\mathcal{A}$ be the finite set of such names. Processes P are defined as: $P ::= \mathbf{0} \mid \alpha.P \mid (P|P) \mid (\nu x)P \mid C(y_1, \cdots, y_n)$. $\mathbf{0}$ refers to the nil process. $\alpha.P$ provides a sequence of operations in the process; first input/output prefix α takes place, and then P executes. $P|P$ provides parallelism in the system. $(\nu x)P$ restricts (binds) name x within P. $C(y_1, \cdots, y_n)$ refers to the invocation of an agent C with parameters $y_1, \cdots, y_n$. Let P, Q, R range over processes.

Free and bound names Name restriction and input prefix bind names in a process. We denote the set of free names in process P with $fn(P)$. α-conversion for bound names is defined in the standard way.

Notational conventions A sequence of names is denoted by $\tilde{a}$, i.e., $a_1, \cdots, a_l$ for some l. A sequence of name restrictions in a process $(\nu a_1)(\nu a_2)\cdots(\nu a_l)P$ is shown by $(\nu a_1 a_2 \cdots a_l)P$, or in short $(\nu\tilde{a})P$. We skip specifying the input name, if it is not free in the following process, i.e., $a.P$ refers to $a(x).P$ where $x \notin fn(P)$. $\bar{a}.P$ refers to outputting a value on link a that can be elided, e.g., due to lack of relevance in discussion.

Codebases Agent definitions are of the form $A(x_1, \cdots, x_n) \triangleq P$. Let's denote the set of agent and sub-agent definitions with $\mathcal{D}$. We assume the existence of a *universal codebase* $\mathcal{C}_\mathcal{U}$ consisting of agent definitions of such form. This codebase is used to define *top-level agents*. A top-level agent corresponds to a concurrent components of the system. Top-level agents are supposed to execute in parallel and occasionally communicate with each other to accomplish their own tasks, and in aggregate the concurrent system. Let $\mathcal{A}_\mathcal{U}$ be the set of top-level agent names such that $\mathcal{A}_\mathcal{U} \subset \mathcal{A}$. Throughout the paper we let m to be the size of $\mathcal{A}_\mathcal{U}$, comprising of $A_1, \cdots, A_m$. $\mathcal{C}_\mathcal{U}$ is defined as a function from top-level agent names to their definitions, i.e., $\mathcal{C}_\mathcal{U} : \mathcal{A}_\mathcal{U} \to \mathcal{D}$.

Moreover, we assume the existence of a *local codebase* for each top-level agent, denoted by $\mathcal{C}_\mathcal{L}(A)$ for top-level agent A. A local codebase consists of sub-agent (subprocess) definitions of the form $B^A(x_1, \cdots, x_n) \triangleq P$, where B is a sub-agent identifier, and A is a top-level agent identifier annotated in the definition of B. We treat sub-agents as internal modules or functions of a top-level process. Annotation of top-level agent identifier is used for this purpose, i.e., B^A specifies that B is a module of top-level agent A. The set of sub-agent names is denoted by $\mathcal{A}_\mathcal{L}$, defined as $\mathcal{A} - \mathcal{A}_\mathcal{U}$. $\mathcal{C}_\mathcal{L}$ is defined as the function with signature $\mathcal{C}_\mathcal{L} : \mathcal{A}_\mathcal{U} \to \mathcal{A}_\mathcal{L} \to \mathcal{D}$.

Note that any process and subprocess definition can be recursive, e.g., if $\mathcal{C}_\mathcal{U}(A) = [A(x_1, \cdots, x_n) \triangleq P]$ then $A(y_1, \cdots, y_n)$ may appear in P. In the following, we use A and B to range over top-level agent identifiers and sub-agent identifiers, resp. We use C to range over both top-level agents, A, and sub-agents, B^A. In the rest of the paper, we refer to top-level agents simply by "agents". We assume that in any (sub)process definition $C(x_1, \cdots, x_n) \triangleq P$, we have $fn(P) \subseteq \{x_1, \cdots, x_n\}$. This ensures that (sub)processes are closed.

Initial system Let $\mathcal{A}_\mathcal{U} = \{A_1, \cdots, A_m\}$. We posit a sequence of links $\tilde{c}$, that connect these agents in the system. Then the initial concurrent system $\mathfrak{s}$ is defined as

$$\mathfrak{s} ::= \langle P_\mathfrak{s}, \mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \rangle, \tag{1}$$

where $P_\mathfrak{s} = (\nu\tilde{c})(A_1(\tilde{x}_1) \mid A_2(\tilde{x}_2) \mid \cdots \mid A_m(\tilde{x}_m))$, assuming $fn(P_\mathfrak{s}) = \emptyset$, i.e., $\bigcup_i \tilde{x}_i \subseteq \tilde{c}$.

Configurations We define system configurations as $\kappa ::= P$, where P is the process associated with the whole system. The initial configuration is then defined as $\kappa_0 = P_\mathfrak{s}$.

Substitutions A substitution is a function $\sigma : \mathcal{N} \to \mathcal{N}$. The notation $\{y/x\}$ is used to refer to a substitution that maps x to y, and acts as the identity function otherwise. $\{\tilde{y}/\tilde{x}\}$ is used to denote multiple explicit mappings in a substitution, where $\tilde{x}$ and $\tilde{y}$ are equal in length. $P\sigma$ refers to replacing free names in P according to σ. This is associated with renaming of bound names in P to avoid name clashes.

3.1.2 Semantics

In the following, we define evaluation contexts and the structural congruence between processes. These definitions facilitate the specification of unlabeled operational semantics in a concise manner.

Evaluation contexts A context is a process with a hole. An evaluation context $\mathcal{E}$ is a context whose hole is not under input/output prefix, i.e., $\mathcal{E} ::= [\,] \mid (\mathcal{E}|P) \mid (P|\mathcal{E}) \mid (\nu a)\mathcal{E}$.

STRUCT
$$\frac{\mathcal{C}_\mathcal{U} \rhd P \equiv P' \qquad \mathcal{C}_\mathcal{U} \rhd Q \equiv Q' \qquad \mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \rhd P \longrightarrow Q}{\mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \rhd P' \longrightarrow Q'}$$

CONTEXT
$$\frac{\mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \rhd P \longrightarrow Q}{\mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \rhd \mathcal{E}[P] \longrightarrow \mathcal{E}[Q]}$$

CALL
$$\frac{\mathcal{C}_\mathcal{L}(A)(B) = [B^A(\tilde{x}) \triangleq P]}{\mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \rhd B^A(\tilde{y}) \longrightarrow P\{\tilde{y}/\tilde{x}\}}$$

COMM
$$\mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \rhd a(x).P \mid \bar{a}b.Q \longrightarrow P\{b/x\} \mid Q$$

Fig. 2. Unlabeled reduction semantics of Π.

Structural congruence Two processes P and Q are structurally congruent under the universal codebase $\mathcal{C}_\mathcal{U}$, denoted by $\mathcal{C}_\mathcal{U} \rhd P \equiv Q$ according to the following rules.

(i) Structural congruence is an equivalence relation.

(ii) Structural congruence is closed by the application of $\mathcal{E}$, i.e., $\mathcal{C}_\mathcal{U} \rhd P \equiv Q$ implies $\mathcal{C}_\mathcal{U} \rhd \mathcal{E}[P] \equiv \mathcal{E}[Q]$.

(iii) If P and Q are α-convertible, then $\mathcal{C}_\mathcal{U} \rhd P \equiv Q$.

(iv) The set of processes is an Abelian semigroup under $\mid$ operator and unit element $\mathbf{0}$, i.e., for any $\mathcal{C}_\mathcal{U}$, P, Q, and R, we have $\mathcal{C}_\mathcal{U} \rhd P|\mathbf{0} \equiv P$, $\mathcal{C}_\mathcal{U} \rhd P|Q \equiv Q|P$, and $\mathcal{C}_\mathcal{U} \rhd P|(Q|R) \equiv (P|Q)|R$.

(v) For all $A \in \mathcal{A}_\mathcal{U}$, if $\mathcal{C}_\mathcal{U}(A) = [A(\tilde{x}) \triangleq P]$, then $\mathcal{C}_\mathcal{U} \rhd A(\tilde{y}) \equiv P\{\tilde{y}/\tilde{x}\}$.

(vi) $\mathcal{C}_\mathcal{U} \rhd (\nu a)\mathbf{0} \equiv \mathbf{0}$.

(vii) If $a \notin fn(P)$, then $\mathcal{C}_\mathcal{U} \rhd (\nu a)(P|Q) \equiv P|(\nu a)Q$.

(viii) $\mathcal{C}_\mathcal{U} \rhd (\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$.

We may elide $\mathcal{C}_\mathcal{U}$ in the specification of the structural congruence, if it is clear from the context.

Operational semantics We define unlabeled reduction system in Figure 2, using judgment $\mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \rhd \kappa \longrightarrow \kappa'$. We may elide $\mathcal{C}_\mathcal{U}$ and $\mathcal{C}_\mathcal{L}$ in the specification of reduction steps, since they are static and may be clear from the context, i.e., $\kappa \longrightarrow \kappa'$.

Note that according to structural congruence rules an agent invocation is structurally congruent to its definition (part v), and thus considered as an "implicit" step of execution according to rule STRUCT. Contrarily, rule CALL defines an "explicit" reduction step for sub-agent invocations. This is due to some technicality in our modeling: invocation of sub-agents could be logging preconditions and/or logging events (introduced in Section 3.2), and hence need special semantic treatment at the time of call (discussed later in Sections 3.3 and 3.4), e.g., deciding whether a record must be stored in the log.

For a (potentially infinite) system trace $\tau = \kappa_0\kappa_1\cdots$, we use notation $\mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \blacktriangleright \tau$ to specify the generation of trace τ under the universal and local codebases $\mathcal{C}_\mathcal{U}$ and $\mathcal{C}_\mathcal{L}$, and according to the aforementioned unlabeled reduction system, i.e., $\mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \rhd \kappa_i \longrightarrow \kappa_{i+1}$ for all $i \in \{0, 1, \cdots\}$.

For a system trace $\tau = \kappa_0\kappa_1\cdots$, system $\mathfrak{s}$ generates τ, denoted by $\mathfrak{s} \Downarrow \tau$ iff $\mathfrak{s}$ is defined as (1), κ_0 is defined as $P_\mathfrak{s}$, and $\mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \rhd \kappa_i \longrightarrow \kappa_{i+1}$ for all $i \in \{0, 1, \cdots\}$.

***toFOL*(·) instantiation for traces** In order to specify a trace logically, we need to instantiate *toFOL*(·) according to Definition 2.11. We consider the following predicates to logically specify a trace: Comm/3, Call/4, Context/2, UniversalCB/3, and LocalCB/4.[6]

Let $\mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \blacktriangleright \tau$, and $\tau = \kappa_0 \cdots \kappa\kappa' \cdots$. Moreover, let t denote a timing counter. We define a function that logically specifies a configuration within a trace. To this end, let the helper function $\mathit{toFOL}(\kappa, t)$ return the logical specification of κ at time t. Essentially, $\mathit{toFOL}(\kappa, t)$ specifies what the evaluation context and the redex are within κ at time t, defined as follows:

(i) $\mathit{toFOL}(\kappa, t) = \{\mathrm{Comm}(t, a(x).P, \bar{a}b.Q), \mathrm{Context}(t, \mathcal{E})\}$,[7] if $\kappa \equiv \mathcal{E}[a(x).P \mid \bar{a}b.Q]$ and $\kappa' \equiv \mathcal{E}[P\{b/x\} \mid Q]$.

(ii) $\mathit{toFOL}(\kappa, t) = \{\mathrm{Call}(t, A, B, \tilde{y}), \mathrm{Context}(t, \mathcal{E})\}$, if $\kappa \equiv \mathcal{E}[B^A(\tilde{y}))]$ and $\kappa' \equiv \mathcal{E}[P\{\tilde{y}/\tilde{x}\}]$. Note that in $\mathrm{Call}(t, A, B, \tilde{y})$, we treat $\tilde{y}$ as a single list of elements, rather than a sequence of elements passed as parameters to Call, i.e., Call is always a quaternary predicate.

As an example, consider α-converted structurally equivalent processes. Let $\kappa = a(x).(\nu b)\bar{x}b.\mathbf{0}|\bar{a}b.\mathbf{0}$. Since $\kappa \equiv \kappa' = a(x).(\nu d)\bar{x}d.\mathbf{0}|\bar{a}b.\mathbf{0}$, and $\kappa' \longrightarrow (\nu d)\bar{b}d.\mathbf{0}|\mathbf{0}$, we have $\kappa \longrightarrow (\nu d)\bar{b}d.\mathbf{0}|\mathbf{0}$ according to the rule STRUCT in Figure 2. Then, $\mathit{toFOL}(\kappa, t) = \mathit{toFOL}(\kappa', t) = \{\mathrm{Comm}(t, a(x).(\nu d)\bar{x}d.\mathbf{0}, \bar{a}b.\mathbf{0}), \mathrm{Context}(t, [\,])\}$.

Logical specification of universal and local codebases, denoted by $\langle\mathcal{C}_\mathcal{U}\rangle$ and $\langle\mathcal{C}_\mathcal{L}\rangle$ resp., are defined as

(i) $\langle\mathcal{C}_\mathcal{U}\rangle = \{\mathrm{UniversalCB}(A, \tilde{x}, P) \mid \mathcal{C}_\mathcal{U}(A) = [A(\tilde{x}) \triangleq P]\}$

[6] $/n$ refers to the arity of the predicate.

[7] Processes and evaluation contexts appear as predicate arguments in this presentation to boost readability. Note that their syntax can be written as string literals to comply with the syntax of predicate logic.

(ii) $\langle \mathcal{C}_\mathcal{L} \rangle = \{\text{LocalCB}(A, B, \tilde{x}, P) \mid \mathcal{C}_\mathcal{L}(A)(B) = [B^A(\tilde{x}) \triangleq P]\}$

Note that in UniversalCB$(A, \tilde{x}, P)$ and LocalCB$(A, B, \tilde{x}, P)$, $\tilde{x}$ is a single list of elements, rather than a sequence of elements passed as parameters to the predicates, and thus these predicates have fixed arities.

We define logical specification of traces both for finite and infinite cases according to the logical specification of configurations, and universal and local codebases, i.e., using $\mathit{toFOL}(\kappa, t)$, $\langle \mathcal{C}_\mathcal{U} \rangle$, and $\langle \mathcal{C}_\mathcal{L} \rangle$. Let $\mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \blacktriangleright \tau$. If τ is finite, i.e., $\tau = \kappa_0\kappa_1 \cdots \kappa_n$ for some n, then its logical specification is defined as $\mathit{toFOL}(\tau) = \bigcup_{i=0}^{n} \mathit{toFOL}(\kappa_i, i) \bigcup \langle \mathcal{C}_\mathcal{U} \rangle \bigcup \langle \mathcal{C}_\mathcal{L} \rangle$. Otherwise, for infinite trace $\tau = \kappa_0\kappa_1 \cdots$, $\mathit{toFOL}(\tau) = \bigcup_{\tau' \in \mathit{prefix}(\tau)} \mathit{toFOL}(\tau') \bigcup \langle \mathcal{C}_\mathcal{U} \rangle \bigcup \langle \mathcal{C}_\mathcal{L} \rangle$, where $\mathit{toFOL}(\tau') = \bigcup_{i=0}^{n} \mathit{toFOL}(\kappa_i, i)$, for $\tau' = \kappa_0\kappa_1 \cdots \kappa_n$. It is straightforward to show that $\mathit{toFOL}(\tau)$ is injective and monotonically increasing.

3.2 A Class of Logging Specifications

We define the class of logging specifications $\mathcal{LS}_{\mathit{call}}$ that specify temporal relations among module invocations in concurrent systems. $\mathcal{LS}_{\mathit{call}}$ is the set of all logging specifications LS defined as $\mathit{spec}(\Gamma_G, \{\text{LoggedCall}\})$, where Γ_G is a set of Horn clauses, called *guidelines*, including clauses of the form

$$\begin{aligned}\forall t_0, \cdots, t_n, \mathit{xs}_0, \cdots, \mathit{xs}_n.\text{Call}(t_0, A_0, B_0, \mathit{xs}_0) \bigwedge_{i=1}^{n} \big(\text{Call}(t_i, A_i, B_i, \mathit{xs}_i) \wedge t_i < t_0\big) \wedge \\ \varphi(t_0, \cdots, t_n) \wedge \varphi'(\mathit{xs}_0, \cdots, \mathit{xs}_n) \implies \text{LoggedCall}(A_0, B_0, \mathit{xs}_0),\end{aligned} \tag{2}$$

in which for all $j \in \{0, \cdots, n\}$, $A_j \in \mathcal{A}_\mathcal{U}$, $B_j \in \mathcal{A}_\mathcal{L}$, xs_j is a placeholder for a list of parameters passed to B_j, and Call$(t_j, A_j, B_j, \mathit{xs}_j)$ specifies the event of invoking module (subprocess) B_j by the top-level process A_j at time t_j with parameters xs_j. In (2), $\varphi(t_0, \cdots, t_n)$ is assumed to be a possibly empty conjunctive sequence of literals of the form $t_i < t_j$. Moreover, we define *triggers* and *logging events* as $\mathit{Triggers}(\mathit{LS}) = \{(A_1, B_1), \cdots, (A_n, B_n)\}$ and $\mathit{Logevent}(\mathit{LS}) = (A_0, B_0)$, resp. *Logging preconditions* are predicates Call$(t_i, A_i, B_i, \tilde{x})$ for all $i \in \{1, \cdots, n\}$. As an additional condition, we assume that $\mathit{Logevent}(\mathit{LS}) \notin \mathit{Triggers}(\mathit{LS})$.

Example 3.1 describes the logging specification for the breaking the glass policy specified in Section 1.2 for a medical records system, using (2).

Example 3.1 We revisit the example described in Section 1.2, a microservices-based medical records system, where breaking the glass entails logging the attempts to read patient medical history. Each microservice is treated as an agent, i.e., agents `Patient` and `Auth` correspond to Patient and Authorization microservices, resp. Let's assume that a user can break the glass by invoking `brkGlass` function from `Auth` agent. Moreover, reading patient medical history is accomplished by calling function `getMedHist` deployed by `Patient` agent. Indeed, these functions are treated as sub-agents in our calculus, i.e., we assume the existence of definitions:

- $\mathcal{C}_\mathcal{L}(\texttt{Auth})(\texttt{brkGlass}) = [\texttt{brkGlass}^{\texttt{Auth}}(u) \triangleq P]$ for some process P, and
- $\mathcal{C}_\mathcal{L}(\texttt{Patient})(\texttt{getMedHist}) = [\texttt{getMedHist}^{\texttt{Patient}}(p, u) \triangleq Q]$ for some process Q.

Then, the logging specification for the breaking-the-glass auditing policy is $\mathit{LS} = \mathit{spec}(\Gamma_G, \{\text{LoggedCall}\})$, where Γ_G includes the clause

$$\begin{aligned}\forall t_0, t_1, p, u.\text{Call}(t_0, \texttt{Patient}, \texttt{getMedHist}, [p, u]) \wedge \text{Call}(t_1, \texttt{Auth}, \texttt{brkGlass}, [u]) \wedge t_1 < t_0 \implies \\ \text{LoggedCall}(\texttt{Patient}, \texttt{getMedHist}, [p, u]).\end{aligned}$$

In the clause above, t_0 and t_1 are timestamps where t_1 is preceding t_0. p refers to the patient identifier whose medical history is requested. u is the healthcare provider (user) identifier who breaks the glass and then attempts to read the medical history of patient p. Note that u is passed as an additional parameter to both `brkGlass` and `getMedHist`. In practice, this is accomplished in microservices using access tokens. An API gateway uses access tokens to communicate the identity of the service requester. One common approach to implement access tokens is by JSON Web Tokens standard [26].

3.3 Target System Model

We define the target system model, denoted by $\Pi_{\log}$, as an extension to Π with the following syntax and semantics. The instrumentation algorithm's job is to map a system specified in Π to a system in $\Pi_{\log}$.

3.3.1 Syntax

$\Pi_{\log}$ extends prefixes with $\alpha ::= \cdots \mid \mathsf{callEvent}(A, B, \tilde{x}) \mid \mathsf{emit}(A, B, \tilde{x}) \mid \mathsf{addPrecond}(x, A) \mid \mathsf{sendPrecond}(x, A)$. $\tilde{x}$ is considered as a single list of names in $\mathsf{callEvent}$ and emit, so that they have fixed arities.

PI
$$\frac{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd P \longrightarrow Q}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd (t, P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, Q, \Delta, \Sigma, \Lambda)}$$

CALL_EV
$$\frac{\Delta'(A) = \Delta(A) \cup \{\mathrm{Call}(t, A, B, \tilde{x})\} \qquad \forall A' \in \mathcal{A}_{\mathcal{U}} - \{A\}.\Delta'(A') = \Delta(A')}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd \quad (t, \mathsf{callEvent}(A, B, \tilde{x}).P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, P, \Delta', \Sigma, \Lambda)}$$

ADD_PRECOND
$$\frac{\Sigma'(A) = \Sigma(A) \cup \{x\} \qquad \forall A' \in \mathcal{A}_{\mathcal{U}} - \{A\}.\Sigma'(A') = \Sigma(A')}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd \quad (t, \mathsf{addPrecond}(x, A).P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, P, \Delta, \Sigma', \Lambda)}$$

SEND_PRECOND
$$\frac{y = \mathtt{serialize}(\Delta(A))}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd \quad (t, \mathsf{sendPrecond}(x, A).P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, \bar{x}y.P, \Delta, \Sigma, \Lambda)}$$

LOG
$$\frac{\Sigma(A) \cup \Delta(A) \cup \Gamma_G \vdash \mathrm{LoggedCall}(A, B, \tilde{x}) \qquad \Lambda'(A) = \Lambda(A) \cup \{\mathrm{LoggedCall}(A, B, \tilde{x})\} \qquad \forall A' \in \mathcal{A}_{\mathcal{U}} - \{A\}.\Lambda'(A') = \Lambda(A')}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd (t, \mathsf{emit}(A, B, \tilde{x}).P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, P, \Delta, \Sigma, \Lambda')}$$

NO_LOG
$$\frac{\Sigma(A) \cup \Delta(A) \cup \Gamma_G \nvdash \mathrm{LoggedCall}(A, B, \tilde{x})}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd (t, \mathsf{emit}(A, B, \tilde{x}).P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, P, \Delta, \Sigma, \Lambda)}$$

Fig. 3. Unlabeled reduction semantics of $\Pi_{\log}$.

A configuration κ, in $\Pi_{\log}$, is defined as the quintuple $\kappa ::= (t, P, \Delta, \Sigma, \Lambda)$, with the following details. t is a timing counter. P is the process associated with the whole concurrent system. Processes in $\Pi_{\log}$ are defined similar to Π, without any extensions. $\Delta(\cdot)$ is a mapping that receives an agent identifier A and returns the set of logical preconditions (to log) that denote the events transpired locally in that agent. That is, $\Delta(A)$ is a set of predicates of the form $\mathrm{Call}(t, A, B, \tilde{x})$. $\Sigma(\cdot)$ is a mapping that receives an agent identifier A and returns the set of all logical preconditions that have taken place in the triggers, i.e., in all agents $A' \in \mathcal{A}_{\mathcal{U}}$, where $(A', B) \in \mathit{Triggers}$ for some $B \in \mathcal{A}_{\mathcal{L}}$. That is, $\Sigma(A)$ is a set of predicates of the form $\mathrm{Call}(t, A', B, \tilde{x})$, where $(A', B) \in \mathit{Triggers}$. These preconditions are supposed to be gathered by A from other agents A', in order to decide whether to log an event. $\Lambda(\cdot)$ is a mapping that receives an agent identifier A and returns the audit log recorded by that agent. $\Lambda(A)$ is a set of predicates of the form $\mathrm{LoggedCall}(A, B, \tilde{x})$. The initial configuration is $\kappa_0 = (0, P_{\mathfrak{s}}, \Delta_0, \Sigma_0, \Lambda_0)$, where for any $A \in \mathcal{C}_{\mathcal{U}}$, $\Delta_0(A) = \Sigma_0(A) = \Lambda_0(A) = \emptyset$.

3.3.2 Semantics

We use judgment $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd \kappa \longrightarrow \kappa'$ to specify a step of reduction in $\Pi_{\log}$. Figure 3 depicts the unlabeled reduction semantics of $\Pi_{\log}$. $\mathcal{C}_{\mathcal{U}}$, $\mathcal{C}_{\mathcal{L}}$, and Γ_G may be elided in the specification of reduction steps since they are static and may be clear from the context.

$\Pi_{\log}$ inherits the reduction semantics of Π, according to rule PI. Rule CALL_EV gives the reduction with prefix $\mathsf{callEvent}(A, B, \tilde{x})$. In this case, Δ gets updated for agent A with information about the invocation of subprocess B^A. In rule ADD_PRECOND, reduction with the prefix $\mathsf{addPrecond}(x, A)$ is specified. In this case, x is added to Σ. Rule SEND_PRECOND is about the reduction with prefix $\mathsf{sendPrecond}(x, A)$. In this case, the set of logging preconditions that are collected by A, i.e., $\Delta(A)$, is converted to a transferable object (aka object serialization), e.g., a string of characters describing the content of $\Delta(A)$, and sent though link x. Let `serialize()` be the semantic function that handles this conversion. With prefix $\mathsf{emit}(A, B, \tilde{x})$, agent A is supposed to study whether the predicate $\mathrm{LoggedCall}(A, B, \tilde{x})$ is logically derivable from the local set of preconditions, i.e., $\Delta(A)$, the set of preconditions that are collected by other agents involved in the enforcement of the logging specification, i.e., $\Sigma(A)$, and the set of guidelines Γ_G. If the predicate is derivable, then it is added to the audit log of A, i.e., $\Lambda(A)$. Otherwise, the log does not change. Rule LOG specifies the former case, whereas the rule NO_LOG specifies the latter.

For a (potentially infinite) system trace $\tau = \kappa_0\kappa_1 \cdots$, we use notation $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \blacktriangleright \tau$ to specify the generation of trace τ under the universal codebase $\mathcal{C}_{\mathcal{U}}$, local codebase $\mathcal{C}_{\mathcal{L}}$, and set of guidelines Γ_G, according to the reduction system, i.e., $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd \kappa_i \longrightarrow \kappa_{i+1}$ for all $i \in \{0, 1, \cdots\}$.

The generated trace in $\Pi_{\log}$ out of a target system $\mathfrak{s}$, i.e., $\mathfrak{s} \Downarrow \tau$, can be defined in the same style as defined in Π, i.e., by some valid initial system in $\Pi_{\log}$ [8], the initial configuration κ_0 in $\Pi_{\log}$, and the aforementioned reduction system for $\Pi_{\log}$.

The residual log of a configuration is defined as $\mathit{logof}(\kappa) = \mathbb{L} = \bigcup_{A \in \mathcal{A}_{\mathcal{U}}} \Lambda(A)$, where $\kappa = (\_, \_, \_, \_, \Lambda)$. [9] This instantiates $\tau \rightsquigarrow \mathbb{L}$ for $\Pi_{\log}$ (Definition 2.4). Since $\mathbb{L}$ is a set of logical literals, it suffices to define $\mathit{toFOL}(\cdot)$ for audit logs as $\mathit{toFOL}(\mathbb{L}) = \mathbb{L}$, which completes the instantiation of $\lfloor \mathbb{L} \rfloor$ (Definition 2.11).

[8] In Section 3.4 one such initial system is given by the instrumentation algorithm.

[9] Underscore is used as wildcard.

Note that arbitrary systems in $\Pi_{\log}$ do not guarantee any correctness of audit logging. However, there is a subset of systems in $\Pi_{\log}$ that provably satisfy this property. These systems use the extended prefixes (introduced as part of $\Pi_{\log}$ syntax) in a particular way for this purpose. In the following section, we introduce an instrumentation algorithm to map any system in Π to a system in $\Pi_{\log}$, and later prove that any instrumented system satisfies correctness results for audit logging.

3.4 Instrumentation Algorithm

Instrumentation algorithm $\mathcal{I}$ takes a Π system, defined in (1), and a logging specification $LS \in \mathcal{LS}_{call}$, defined in Section 3.2, and produces a system $\mathfrak{s}'$ in $\Pi_{\log}$ defined as $\mathfrak{s}' = \langle P'_{\mathfrak{s}}, \mathcal{C}'_{\mathcal{U}}, \mathcal{C}'_{\mathcal{L}} \rangle$, where $P'_{\mathfrak{s}} = (\nu\tilde{c})(\nu\tilde{c}')(A_1(\tilde{x}'_1) \mid A_2(\tilde{x}'_2) \mid \cdots \mid A_m(\tilde{x}'_m))$. $\tilde{c}'$ is the sequence of names of the form c_{ij} which are all fresh, i.e., they are not used already in (1). Moreover, it is assumed that sub-agent identifiers D_{ij} are also fresh, i.e., they are undefined in $\mathcal{C}_{\mathcal{L}}$ component of (1).

Intuitively, $\mathcal{I}$ works as follows.

(i) $\mathcal{I}$ adds new links c_{ij} between agents A_i and A_j, where A_i is the agent that includes a sub-agent whose invocation is considered a logging event, and A_j is some agent that includes a sub-agent whose invocation is a trigger for that logging event. c_{ij} is used as a link between A_i and A_j to communicate logging preconditions (by sendPrecond and addPrecond prefixes).

(ii) Regarding the invocation of a sub-agent B^A,
   (a) if the invocation of B^A is a trigger, then the execution of B^A must be preceded by callEvent prefix. This way, the invocation of B^A is stored in A's local set of logging precondition ($\Delta(A)$), according to the rule CALL_EV.
   (b) if the invocation of B^A is a logging event, then execution of B^A must be preceded by callEvent, similar to the case above. Next, it must communicate on appropriate links (c_{ij}s) with all other agents that are involved as triggers according to the logging specification. To this end, B^A is supposed to notify each of those agents to send their collected preconditions. After receiving all those preconditions from involved agents on the dedicated links, it adds them to $\Sigma(A)$. This is done using addPrecond prefixes, according to the rule ADD_PRECOND. Then, it studies whether the invocation must be logged, before following normal execution. This is facilitated by emit prefix (rules LOG and NO_LOG).
   (c) if the invocation of B^A is neither a trigger nor a logging event, then that sub-agent executes without any change in behavior.

(iii) Regarding the invocation of an agent A
   (a) if A includes a sub-agent B^A whose invocation is considered a trigger, then A must be able to receive and handle incoming requests for collected preconditions. This is done by adding a subprocess to A that always listens for requests on the dedicated link (c_{ij}) between itself and the agent that may send such requests. Upon receiving such a request, it sends back the preconditions, handled by prefix sendPrecond according to the rule SEND_PRECOND, and then continues to listen on the link.
   (b) if A does not include any trigger invocation of a sub-agent, then A executes without any changes.

Formally, the details of the returned system $\mathfrak{s}'$ are as follows:

(i) $\tilde{c}'$: is the sequence of all names c_{ij} where $(A_i, B) = \mathit{Logevent}(LS)$ for some $B \in \mathcal{A}_{\mathcal{L}}$, and $(A_j, B') \in \mathit{Triggers}(LS)$ for some $B' \in \mathcal{A}_{\mathcal{L}}$.

(ii) $\mathcal{C}'_{\mathcal{L}}$:
   (a) $\mathcal{C}'_{\mathcal{L}}(A)(B) = [B^A(\tilde{x}) \triangleq \mathsf{callEvent}(A, B, \tilde{x}).P]$, if $(A, B) \in \mathit{Triggers}(LS)$ and $\mathcal{C}_{\mathcal{L}}(A)(B) = [B^A(\tilde{x}) \triangleq P]$.
   (b) $\mathcal{C}'_{\mathcal{L}}(A_0)(B_0) = [B_0^{A_0}(\tilde{x}) \triangleq \mathsf{callEvent}(A_0, B_0, \tilde{x}) \,.\, \bar{c_{01}} \,.\, \cdots \,.\, \bar{c_{0n}} \,.\, c_{01}(p_1) \,.\, \cdots \,.\, c_{0n}(p_n) \,.\, \mathsf{addPrecond}(p_1, A_0) \,.\, \cdots \,.\, \mathsf{addPrecond}(p_n, A_0) \,.\, \mathsf{emit}(A_0, B_0, \tilde{x}).P]$, if $(A_0, B_0) = \mathit{Logevent}(LS)$, $\mathcal{C}_{\mathcal{L}}(A_0)(B_0) = [B_0^{A_0}(\tilde{x}) \triangleq P]$, and $\mathit{Triggers}(LS) = \{(A_1, B_1), \cdots, (A_n, B_n)\}$.
   (c) $\mathcal{C}'_{\mathcal{L}}(A)(B) = \mathcal{C}_{\mathcal{L}}(A)(B)$, otherwise.

(iii) $\mathcal{C}'_{\mathcal{U}}$:
   (a) If $(A_j, B) \in \mathit{Triggers}(LS)$ for some $B \in \mathcal{A}_{\mathcal{L}}$, and A_i be the agent that $(A_i, B') = \mathit{Logevent}(LS)$ for some $B' \in \mathcal{A}_{\mathcal{L}}$, then $\mathcal{C}'_{\mathcal{U}}(A_j) = [A_j(\tilde{x}, c_{ij}) \triangleq P | D_{ij}^{A_j}(c_{ij})]$, where $\mathcal{C}_{\mathcal{U}}(A_j) = [A_j(\tilde{x}) \triangleq P]$, and $\mathcal{C}'_{\mathcal{L}}(A_j)(D_{ij}) = [D_{ij}^{A_j}(c_{ij}) \triangleq c_{ij}.\mathsf{sendPrecond}(c_{ij}, A_j).D_{ij}^{A_j}(c_{ij})]$.
   (b) If $(A_j, B) \notin \mathit{Triggers}(LS)$ for any $B \in \mathcal{A}_{\mathcal{L}}$, then $\mathcal{C}'_{\mathcal{U}}(A_j) = \mathcal{C}_{\mathcal{U}}(A_j)$.

Note that D_{ij} is defined recursively to facilitate listening on c_{ij} indefinitely for incoming requests about logging preconditions. In addition, since c_{ij} is fresh, $c_{ij} \notin \mathit{fn}(P)$. Therefore, P cannot communicate on this link, e.g., to compromise logging attempts. Figure 4 illustrates the established links between sub-agents of different

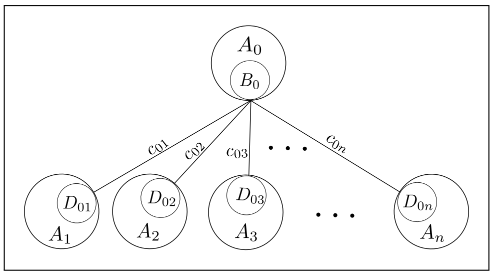


Fig. 4. Illustration of the established links in the instrumented concurrent system.

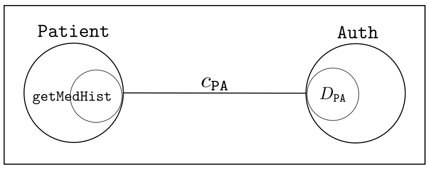


Fig. 5. Example: Illustration of the established link in the instrumented medical records system.

$$trim(\mathbf{0}) = \mathbf{0} \qquad trim(\alpha.P) = \begin{cases} trim(P) & \text{if } \alpha = c_{ij}, \bar{c_{ij}}, c_{ij}(x), \bar{c_{ij}}x, \\ & \mathsf{callEvent}(A, B, \tilde{x}), \mathsf{addPrecond}(x, A), \\ & \mathsf{sendPrecond}(x, A), \mathsf{emit}(A, B, \tilde{x}) \\ \alpha.trim(P) & \text{otherwise} \end{cases}$$

$$trim(P|Q) = \begin{cases} trim(P) & \text{if } Q = D^A_{ij} \text{ for some } i, j, A \\ trim(Q) & \text{if } P = D^A_{ij} \text{ for some } i, j, A \\ trim(P)|trim(Q) & \text{otherwise} \end{cases} \qquad trim((\nu x)P) = \begin{cases} trim(P) & \text{if } x = c_{ij} \text{ for some } i, j \\ (\nu x)trim(P) & \text{otherwise} \end{cases}$$

$$trim(C(\tilde{x})) = \begin{cases} trim(C(\tilde{y})) & \text{if } \tilde{x} = \tilde{y}, c_{ij} \text{ for some } i, j \\ C(\tilde{x}) & \text{otherwise} \end{cases}$$

Fig. 6. Function *trim*.

agents according to the guideline defined in (2). These links are used to communicate logging preconditions between the logging event and the triggers.

Example 3.2 depicts how the medical records system described in Section 1.2 is instrumented according to the specified instrumentation algorithm, and the logging specification given in Example 3.1.

Example 3.2 In Example 3.1, $(\texttt{Patient}, \texttt{getMedHist})$ is the logging event, and $\{(\texttt{Auth}, \texttt{brkGlass})\}$ is the set of triggers. Applying the instrumentation algorithm changes the systems as follows. A new link $c_{\texttt{PA}}$ is established between `Patient` and `Auth` agents. The definition of `brkGlass` is updated as $\mathcal{C}'_{\mathcal{L}}(\texttt{Auth})(\texttt{brkGlass}) = [\texttt{brkGlass}^{\texttt{Auth}}(u) \triangleq \mathsf{callEvent}(\texttt{Auth}, \texttt{brkGlass}, [u]).P]$, and the definition of `getMedHist` is updated as

$$\mathcal{C}'_{\mathcal{L}}(\texttt{Patient})(\texttt{getMedHist}) = [\texttt{getMedHist}^{\texttt{Patient}}(p, u) \triangleq \mathsf{callEvent}(\texttt{Patient}, \texttt{getMedHist}, [p, u]).\bar{c_{\texttt{PA}}}.c_{\texttt{PA}}(f). \\ \mathsf{addPrecond}(f, \texttt{Patient}).\mathsf{emit}(\texttt{Patient}, \texttt{getMedHist}, [p, u]).Q].$$

In addition subprocess $D_{\texttt{PA}}$ is added to agent `Auth` that indefinitely responds to the requests from `Patient` on link $c_{\texttt{PA}}$, defined as: $\mathcal{C}'_{\mathcal{L}}(\texttt{Auth})(D_{\texttt{PA}}) = [D^{\texttt{Auth}}_{\texttt{PA}}(c_{\texttt{PA}}) \triangleq c_{\texttt{PA}}.\mathsf{sendPrecond}(c_{\texttt{PA}}, \texttt{Auth}).D^{\texttt{Auth}}_{\texttt{PA}}(c_{\texttt{PA}})]$. Figure 5 illustrates the established link between the sub-agents of the two agents.

3.4.1 Instantiation of $:\approx$

According to Definition 2.5, semantics preservation relies on an abstraction of correspondence relation $:\approx$ between source and target traces. In this section, we instantiate this relation for $\mathcal{I}$. We define the source and target trace correspondence relation as follows: $\tau_1\kappa_1 :\approx \tau_2\kappa_2$ iff $\kappa_1 = P_1$, $\kappa_2 = (t_2, P_2, \Delta_2, \Sigma_2, \Lambda_2)$, and $trim(P_2) = P_1$. Function *trim* is formally defined in Figure 6. Intuitively, it removes all prefixes, sub-agents, and link names that $\mathcal{I}$ may add to a process.

3.4.2 Main Results

Main properties include three results. The instrumentation algorithm $\mathcal{I}$ is semantics preserving, deadlock-free, correct. These are specified in Theorems 3.3, 3.4, and 3.5, resp. Proofs of the theorems are given in our accompanying Technical Report [3].

Theorem 3.3 (Semantics preservation) *$\mathcal{I}$ is semantics preserving (Definition 2.5).*

Theorem 3.4 (Deadlock-freeness) *$\mathcal{I}$ is deadlock-free (Definition 2.6).*

Theorem 3.5 (Instrumentation correctness) *$\mathcal{I}$ is correct (Definition 2.7).*

4 Related Work

Majority of previous work on audit logging in concurrent environments focus on audit log analysis (e.g., [37]) and security concerns regarding in transit and/or at rest log information (e.g., [30,57,1]). Studies regarding the collection of logs from multiple monitors in distributed intrusion detection systems are such instances (e.g., [54,58]). However, previous work do not reflect on the generation of the log, and assume that the audit log is given. This line of work includes studies on the security of audit logs in terms of their secrecy and integrity within concurrent environments. For example, Yavuz et al. [57] propose a logging scheme that guarantees forward security, employing cryptographic techniques. In the same line of work, Böck et al. [10] propose a system that ensures that logs are trustworthy. Our work is however orthogonal to the notion of audit log security. Using cryptographic techniques to ensure verifiable confidentiality and integrity of the audit log does not guarantee it to be correct. We employ a semantic framework by which the content of the audit log can be judged against the execution trace of the concurrent system given in π calculus (Definition 2.3).

Cederquist et al. [13] and Corin et al. [14] propose predicate logic frameworks to specify and enforce accountability requirements in distributed systems. The former proposes a framework that ensures user accountability in discretionary access control. The latter studies user accountability in access to personal data that are associated with usage policies defined by the owner of data, and can be distributed among users. Jagadeesan et al. [24] use turn-based games to analyze distributed accountability systems. Guts. et al. [22] use static type enforcement to assure that a distributed system generates sufficient audit logs. However, our approach is dynamic and relies on instrumentation techniques that can be applied to legacy systems which may inherently suffer from the lack of correct audit logging mechanisms. With respect to system instrumentation for auditing purposes, our work is related to the language proposed by Martin et al. [31] that facilitates querying runtime behavior of a program.

Another line of work employs logs to record proof of legitimate access to system resources. Vaughn et al. [48] propose an architecture based on trusted kernels that rely on such logged proofs. Another related work is the a posteriori compliance control system [19] that verifies legitimacy of access after the fact, using a trust-based logical framework that focuses on a limited set of operations. However, our logical framework is used to specify invocation of any arbitrary operation as a precondition to log, or the logging event.

Audit logs can be considered a form of provenance [41]. CamFlow [39] is an auditing and provenance capture utility in Linux that can easily integrate with distributed systems. Pasquier et al. [40] make strong case for accountability, data provenance and audit in the IoT. AccessProv [12] is proposed as an instrumentation tool that rewrites legacy Java applications for provenance and finds bugs in authorization systems. Kacianka et al. [27] propose a formal model of accountability for cyber-physical systems.

Amir-Mohammadian et al. [6] propose a semantic framework for audit logging based on the theory of information algebra [28,29]. Their implementation model is restricted to sequential computation. This model is therefore insufficient to apply on concurrent systems where logging preconditions and logging events may transpire in different execution threads. Moreover, their implementation model is restricted to deterministic system behavior. Our work generalizes the application of information-algebraic semantics of audit logging to concurrent environments, which naturally behave non-deterministically at runtime. We show that the semantic framework is inclusive enough for this purpose. Similar to [6], we propose a provably correct instrumentation algorithm. However, our algorithm retrofits a concurrent system (rather than a simple sequential program) according to a formal description of audit logging requirements. Information-algebraic semantic framework for audit logging has also been used to enhance dynamic integrity taint analysis through after-the-fact study of audit logs [7,44]. This line of work introduces maybe-tainted tags for data objects and proposes an implementation model on a core functional object-oriented calculus that provably ensures correctness of generated audit logs. However, it does not address the problem of deploying audit logging in concurrent environments.

Recently, Justification Logic [8] is used to formally characterize auditing of computational units [9,42,51] which result in programming languages that enable applications to study their own audit trails and decide accordingly. This is a separate theoretical problem than what we are aiming in this work.

Microservices-based approach [17,20] to software deployment is an application of our implementation model, that we aim to study in future in a greater detail. Accountability plays a significant role as part of the access control framework in microservices-based systems [33], including platform-specific monitoring techniques, e.g., in Azure Kubernetes Service [52]. Smith et al. [45] have proposed a provenance management system, including provenance logger, for microservices-based applications. Camilli et al. [11] have proposed a semantics for microservices based on Petri nets. Our approach is, however, language-based and relies on process calculi. Jolie [25] is the major programming language for the deployment of microservices, whose semantics [21,36] is defined as a process calculus, heavily influenced by π-calculus.

5 Future Work and Conclusion

In this paper, we have proposed an implementation model to enforce correct audit logging in concurrent environments. In essence, we have proposed an algorithm that instruments legacy concurrent systems according to a formal specification of audit logging requirements. We use Horn clause logic to specify these logging requirements, which assert temporal relations among the events that transpire in different concurrent components of the system. We have proven that our algorithm is semantics preserving, i.e., the instrumented system behaves similar to the original system, modulo operations that correspond to audit logging. Moreover, we have proven that our algorithm guarantees correct audit logs. This ensures that the instrumented system avoids missing any logging event, as well as logging unnecessary events. Correctness of audit logs are defined according to an information-algebraic semantic framework. In this semantic framework, information containment is used to compare the runtime behavior vs. the generated audit log.

We have argued that the our instrumentation algorithm proposes a model to implement audit logging in real concurrent systems, e.g., in microservices-based medical records systems (Examples 3.1 and 3.2). In this paper, we have aimed at the formal specification of the model on an abstract core calculus to demonstrate the main ideas for future deployment. In future work, we intend to consider real-world language settings, relying on the fundamental results established in this work. In particular, we are aiming to deploy our existing instrumentation algorithm in Spring Boot [53], a Java microservices framework. Indeed, to provide formal guarantees of audit logging correctness in such real-world settings that are implemented using our model, translation of systems to Π systems is required.

Another area of interest is to extend the class of logging specifications, and hence implementation models. Our current class focuses on function invocations within each agent of the system, and it is limited to Horn clauses. While a great percentage of system events can be specified in this class, we need other classes of logging specifications for certain purposes. For example, consider the effect of revoking break-the-glass status for a user on the specification of audit logging requirements. Moreover, auditing usually includes the log of message transmissions between specific agents, which is not supported by what we have introduced in this paper.

Since our model of concurrency is based on a process calculi, message-passing is used for IPC. This necessitates the exploration of models that study the specification and enforcement of correct audit logging in concurrent environments which handle IPC through alternative approaches, e.g., shared memory and/or files.

References

[1] Accorsi, R., *Bbox: A distributed secure log architecture*, in: *Public Key Infrastructures, Services and Applications - 7th European Workshop, EuroPKI 2010, Athens, Greece, September 23-24, 2010. Revised Selected Papers*, 2010, pp. 109–124.

[2] Allan, C., P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam and J. Tibble, *Adding trace matching with free variables to AspectJ*, in: *OOPSLA 2005*, 2005, pp. 345–364.

[3] Amir-Mohammadiah, S. and C. Kari, *Correct audit logging in concurrent systems (technical report)*, Technical report, University of the Pacific (2020).

[4] Amir-Mohammadian, S., "A Formal Approach to Combining Prospective and Retrospective Security," Ph.D. thesis, The University of Vermont (2017).

[5] Amir-Mohammadian, S., S. Chong and C. Skalka, *Foundations for auditing assurance*, in: *Layered Assurance Workshop*, 2015.

[6] Amir-Mohammadian, S., S. Chong and C. Skalka, *Correct audit logging: Theory and practice*, in: *Principals of Security and Trust*, 2016, pp. 139–162.

[7] Amir-Mohammadian, S. and C. Skalka, *In-depth enforcement of dynamic integrity taint analysis*, in: *Programming Languages and Analysis for Security*, 2016.

[8] Artemov, S., *Justification logic*, in: *European Workshop on Logics in Artificial Intelligence*, Springer, 2008, pp. 1–4.

[9] Bavera, F. and E. Bonelli, *Justification logic and audited computation*, Journal of Logic and Computation **28** (2015), pp. 909–934.

[10] Böck, B., D. Huemer and A. M. Tjoa, *Towards more trustable log files for digital forensics by means of "trusted computing"*, in: *AINA 2010* (2010), pp. 1020–1027.

[11] Camilli, M., C. Bellettini, L. Capra and M. Monga, *A formal framework for specifying and verifying microservices based process flows*, in: *International Conference on Software Engineering and Formal Methods*, Springer, 2017, pp. 187–202.

[12] Capobianco, F., C. Skalka and T. Jaeger, *ACCESSPROV: Tracking the provenance of access control decisions*, in: *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017)*, 2017.

[13] Cederquist, J. G., R. Corin, M. A. C. Dekker, S. Etalle, J. I. den Hartog and G. Lenzini, *Audit-based compliance control*, International Journal of Information Security **6** (2007), pp. 133–151.

[14] Corin, R., S. Etalle, J. I. den Hartog, G. Lenzini and I. Staicu, *A logic for auditing accountability in decentralized systems*, in: *FAST 2004*, 2004, pp. 187–201.

[15] DeYoung, H., D. Garg, L. Jia, D. Kaynar and A. Datta, *Privacy policy specification and audit in a fixed-point logic: How to enforce HIPAA, GLBA, and all that*, Technical Report CMU-CyLab-10-008, Carnegie Mellon University (2010).

[16] DeYoung, H., D. Garg, L. Jia, D. K. Kaynar and A. Datta, *Experiences in the logical specification of the HIPAA and GLBA privacy laws*, in: *WPES 2010*, 2010, pp. 73–82.

[17] Dragoni, N., S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin and L. Safina, *Microservices: yesterday, today, and tomorrow*, in: *Present and ulterior software engineering*, Springer, 2017 pp. 195–216.

[18] Erlingsson, Ú., "The inlined reference monitor approach to security policy enforcement," Ph.D. thesis, Cornell University (2003).

[19] Etalle, S. and W. H. Winsborough, *A posteriori compliance control*, in: *SACMAT 2007*, 2007, pp. 11–20.

[20] Guidi, C., I. Lanese, M. Mazzara and F. Montesi, *Microservices: a language-based approach*, in: *Present and Ulterior Software Engineering*, Springer, 2017 pp. 217–225.

[21] Guidi, C., R. Lucchi, R. Gorrieri, N. Busi and G. Zavattaro, *Sock: a calculus for service oriented computing*, in: *International Conference on Service-Oriented Computing*, Springer, 2006, pp. 327–338.

[22] Guts, N., C. Fournet and F. Z. Nardelli, *Reliable evidence: Auditability by typing*, in: *European Symposium on Research in Computer Security*, Springer, 2009, pp. 168–183.

[23] Hoare, C. A. R., *Communicating sequential processes*, in: *The origin of concurrent programming*, Springer, 1978 pp. 413–443.

[24] Jagadeesan, R., A. Jeffrey, C. Pitcher and J. Riely, *Towards a theory of accountability and audit*, in: *ESORICS 2009*, 2009, pp. 152–167.

[25] *Jolie Programming Language*, https://www.jolie-lang.org/, accessed: 2019-09-18.

[26] *Introduction to JSON Web Tokens*, https://jwt.io/introduction/ (2019), accessed: 2019-09-02.

[27] Kacianka, S. and A. Pretschner, *Understanding and formalizing accountability for cyber-physical systems*, in: *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, IEEE, 2018, pp. 3165–3170.

[28] Kohlas, J., "Information Algebras: Generic Structures For Inference," Discrete mathematics and theoretical computer science, Springer, 2003.

[29] Kohlas, J. and J. Schmid, *An algebraic theory of information: An introduction and survey*, Information **5** (2014), pp. 219–254.

[30] Lee, A. J., P. Tabriz and N. Borisov, *A privacy-preserving interdomain audit framework*, in: *Proceedings of the 2006 ACM Workshop on Privacy in the Electronic Society, WPES 2006, Alexandria, VA, USA, October 30, 2006*, 2006, pp. 99–108.

[31] Martin, M., B. Livshits and M. S. Lam, *Finding application errors and security flaws using PQL: A program query language*, in: *OOPSLA 2005* (2005), pp. 365–383.

[32] Matthews, P. and H. Gaebel, *Break the glass*, in: *HIE Topic Series*, Healthcare Information and Management Systems Society, 2009 .

[33] McLarty, M., R. Wilson and S. Morrison, "Securing Microservice APIs," O'Reilly Media, Inc., 2018.

[34] Milner, R., "Communication and concurrency," Prentice hall New York etc., 1989.

[35] Milner, R., "Communicating and mobile systems: the pi calculus," Cambridge university press, 1999.

[36] Montesi, F. and M. Carbone, *Programming services with correlation sets*, in: *International Conference on Service-Oriented Computing*, Springer, 2011, pp. 125–141.

[37] Mounji, A., B. L. Charlier, D. Zampuniéris and N. Habra, *Distributed audit trail analysis*, in: *1995 Symposium on Network and Distributed System Security, (S)NDSS '95, San Diego, California, February 16-17, 1995*, 1995, pp. 102–113.

[38] Parrow, J., *An introduction to the π-calculus*, in: *Handbook of Process Algebra*, Elsevier, 2001 pp. 479–543.

[39] Pasquier, T., X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer and J. Bacon, *Practical whole-system provenance capture*, in: *Proceedings of the 2017 Symposium on Cloud Computing*, ACM, 2017, pp. 405–418.

[40] Pasquier, T., J. Singh, J. Powles, D. Eyers, M. Seltzer and J. Bacon, *Data provenance to audit compliance with privacy policy in the internet of things*, Personal and Ubiquitous Computing **22** (2018), pp. 333–344.

[41] Ricciotti, W., *A core calculus for provenance inspection*, in: *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, ACM, 2017, pp. 187–198.

[42] Ricciotti, W. and J. Cheney, *Strongly normalizing audited computation*, arXiv preprint arXiv:1706.03711 (2017).

[43] Schneider, F. B., *Enforceable security policies*, ACM Transactions on Information and System Security **3** (2000), pp. 30–50.

[44] Skalka, C., S. Amir-Mohammadian and S. Clark, *Maybe tainted data: Theory and a case study*, Journal of Computer Security **28** (2020), pp. 295–335.

[45] Smith, W., T. Moyer and C. Munson, *Curator: provenance management for modern distributed systems*, in: *10th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2018)*, 2018.

[46] *SWI Prolog*, https://www.swi-prolog.org/, accessed: 2019-09-15.

[47] Staicu, C., D. Schoepe, M. Balliu, M. Pradel and A. Sabelfeld, *An empirical study of information flows in real-world javascript*, CoRR **abs/1906.11507** (2019).

[48] Vaughan, J. A., L. Jia, K. Mazurak and S. Zdancewic, *Evidence-based audit*, in: *CSF 2008*, 2008, pp. 177–191.

[49] *Microservices in Healthcare: Granulate to Accelerate*, https://vicert.com/local/resources/assets/pdf/WhitePaper_Microservices%20in%20Healthcare.pdf (2019), accessed: 2020-02-25.

[50] *Top 10-2017 A10-Insufficient Logging & Monitoring*, https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A10-Insufficient_Logging%252526Monitoring (2017), accessed: 2020-06-04.

[51] W. Ricciotti and J. Cheney, *Explicit auditing*, arXiv preprint arXiv:1808.00486 (2018).

[52] Wasson, M., *Monitoring a microservices architecture in Azure Kubernetes Service (AKS)*, https://docs.microsoft.com/en-us/azure/architecture/microservices/logging-monitoring (2019), accessed: 2019-09-15.

[53] Webb, P., D. Syer, J. Long, S. Nicoll, R. Winch, A. Wilkinson, M. Overdijk, C. Dupuis and S. Deleuze, *Spring boot reference guide*, Part IV. Spring Boot features **24** (2013).

[54] Wu, Y., B. Foo, Y. Mei and S. Bagchi, *Collaborative intrusion detection system (CIDS): A framework for accurate and efficient IDS*, in: *19th Annual Computer Security Applications Conference (ACSAC 2003), 8-12 December 2003, Las Vegas, NV, USA*, 2003, pp. 234–244.

[55] *XSB Prolog*, https://xsb.com/xsb-prolog, accessed: 2019-09-15.

[56] Xu, T., H. M. Naing, L. Lu and Y. Zhou, *How do system administrators resolve access-denied issues in the real world?*, in: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ACM, 2017, pp. 348–361.

[57] Yavuz, A. A. and P. Ning, *BAF: an efficient publicly verifiable secure audit logging scheme for distributed systems*, in: *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, 7-11 December 2009*, 2009, pp. 219–228.

[58] Yegneswaran, V., P. Barford and S. Jha, *Global intrusion detection in the DOMINO overlay system*, in: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*, 2004.

[59] *Learning the lessons of the Dixons Carphone breach*, https://www.zdnet.com/article/learning-the-lessons-of-the-dixons-carphone-breach/ (2020), accessed: 2020-01-15.

[60] Zhang, W., Y. Chen, T. Cybulski, D. Fabbri, C. Gunter, P. Lawlor, D. Liebovitz and B. Malin, *Decide now or decide later?: Quantifying the tradeoff between prospective and retrospective access decisions*, in: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2014, pp. 1182–1192.

[61] *Global Microservices In Healthcare Market Will Reach USD 519 Million By 2025*, https://www.globenewswire.com/news-release/2019/03/14/1753060/0/en/Global-Microservices-In-Healthcare-Market-Will-Reach-USD-519-Million-By-2025-Zion-Market-Research.html (2019), accessed: 2019-09-01.

A constraint-based language for multiparty interactions [1]

Linda Brodo[2]

Università degli Studi di Sassari, Italy.

Carlos Olarte[3]

ECT, Universidade Federal do Rio Grande do Norte

Abstract

Multiparty interactions are common place in today's distributed systems. An agent usually communicates, in a single session, with other agents to accomplish a given task. Take for instance an online transaction including the vendor, the client, the credit card system and the bank. When specifying this kind of system, we probably observe a single transaction including several (binary) communications leading to changes in the state of all the involved agents. Multiway synchronization process calculi, that move from a binary to a multiparty synchronization discipline, have been proposed to formally study the behavior of those systems. However, adopting models such as Bodei, Brodo, and Bruni's Core Network Algebra (CNA), where the number of participants in an interaction is not fixed a priori, leads to an exponential blow-up in the number of states/behaviors that can be observed from the system. In this paper we explore mechanisms to tackle this problem. We extend CNA with constraints that declaratively allow the modeler to restrict the interaction that should actually happen. Our extended process algebra, called CCNA, finds application in balancing the interactions in a concurrent system, leading to a simple, deadlock-free and fair solution for the Dinning Philosopher problem. Our definition of constraints is general enough and it offers the possibility of accumulating costs in a multiparty negotiation. Hence, only computations respecting the thresholds imposed by the modeler are observed. We use this machinery to neatly model a Service Level Agreement protocol. We develop the theory of CCNA including its operational semantics and a behavioral equivalence that we prove to be a congruence. We also propose a prototypical implementation that allows us to verify, automatically, some of the systems explored in the paper.

Keywords: Concurrency theory, constraints, multiparty interactions

1 Introduction

Nowadays concurrent and mobile systems are ubiquitous in several domains and applications. They pervade different areas in science (biological and chemical systems), engineering (security protocols and mobile and service oriented computing) and even the arts (tools for multimedia interaction). In general, concurrent systems exhibit complex forms of interaction, not only among their internal components, but also with the surrounding environment. Hence, a legitimate challenge is to provide computational models allowing us to understand the nature and the behavior of such complex systems. As an answer to this challenge, process algebra such as CCS [17], the π-calculus [18] and CSP [13] among several others have arisen as mathematical formalisms to model and reason about concurrent systems.

It is worth noticing that there is no unified model for concurrency, as in the case of the λ-calculus for sequential computations. Concurrency is, in fact, a relatively young area in computer science, and there are many models that accurately capture some behaviors but ignore/abstract some others. For instance, CCS focuses on the synchronization of processes: by exhibiting complementary actions two processes handshake

[1] The joint work of the authors was possible thanks to the Visiting Scientist grant of University of Sassari. Carlos Olarte was also funded by CNPq. Linda Brodo was also funded by "*fondo di Ateneo per la ricerca 2019*" of University of Sassari.

[2] Email: `brodo@uniss.it`

[3] Email: `carlos.olarte@gmail.com`

and synchronize. However, no message is indeed sent among the agents. Data-passing extensions of CCS allow to overcome this problem, but the underlying communication network remains invariant during computation since processes cannot create and communicate new/private communication channels. The π-calculus gives a step forward and allows the communication of names (representing data and also communication links) that can later be used in other interactions. Many other process algebras have emerged as extensions of existing ones to cope with specific behaviors or they have taken inspiration from particular systems as in the case of calculi for system biology. For instance, the Spi Calculus [1] incorporates cryptographic primitives into the π-calculus for the specification and verification of security protocols and the Brane Calculi [10] took inspiration from the interaction of cell's membranes to model biological interactions. The beauty of all these formalisms relies on their simplicity (few operators), formal semantics and reasoning techniques, including behavioral equivalences (e.g., bisimulation), modal logics and model checking for specifying and verifying system's properties.

Most of the process algebras in the literature focus on binary interactions. Consider for instance CCS where a process $a.P$ (P prefixed by the *input* action a) can synchronize with $\overline{a}.Q$ (Q prefixed by the *output* action $\overline{a}$) when they are in a parallel composition (e.g., as in $a.P \mid \overline{a}.Q$). Such synchronization is made explicit via a special action τ (called the silent action) and what we observe is the synchronization of these processes in the transition $a.P \mid \overline{a}.Q \xrightarrow{\tau} P \mid Q$ where, after the handshake, P and Q continue their executions.

In this paper we shall focus on a multiparty extension of CCS, the so called Core Network Algebra (CNA) [4,6,5], a multiparty process algebra where the number of participants in each synchronization is not fixed a priori. In CNA, the binary interaction of CCS is extended and the usual input and output prefixes are generalized to *links*, e.g., ${}^{a}\backslash_{b}$, that can be thought of as the forwarding of a message received on channel a (the input channel) to another channel b (the output channel). The standard input and output prefixes of CCS are recovered when links expose only an output (${}^{\tau}\backslash_{b}$), or an input (${}^{a}\backslash_{\tau}$); these particular actions are the ends of a *link chain* where τ is the *silent* action (as in CCS). A link chain is the mechanism in CNA by which $n \geq 2$ entities can synchronize. Each entity must offer a link that has to match with an adjacent link offered by another entity. For instance, if three processes offer, respectively, the links ${}^{a}\backslash_{b}$, ${}^{b}\backslash_{c}$ and ${}^{c}\backslash_{d}$, they can synchronize and produce the link chain ${}^{a}\backslash_{b}^{b}\backslash_{c}^{c}\backslash_{d}$, where information flows from a to d through b and c.

In [7,8] the authors showed that the multiparty and open (arbitrary number of participants) nature of CNA poses interesting problems for the point of view of verification. In particular, while the number of possible successor states from a CCS process is quadratic on the number of its outermost prefixes, it is exponential in the case of a CNA process. Moreover, it is possible to specify a graph of agents as a CNA process P (some examples in Section 4.2) and we can check that there is a Hamiltonian path in the graph iff there is exactly one immediate successor of P. We explored in [7,8] symbolic techniques aiming to tame the inherent complexity of the CNA transition system.

The goal of the present paper is to define a suitable extension of CNA that allows the specifier to control, in a declarative way, the behavior of processes. More precisely, we provide mechanisms that allow us to include in a CNA specification certain restrictions that appear naturally in the modeled system at hand. For instance, we may be interested in transitions/paths in a graph to have certain length or to specify upper bounds on the number of participants in an open interaction. This is the purpose of introducing constrains and values in prefixes: the process ${}^{a}\backslash_{b}\langle !v_p\rangle(?c_p).P$ offers the link ${}^{a}\backslash_{b}$ at a given cost/value v_p and checks whether the interaction with some other processes satisfies the constraint c_p. When this process interacts with, e.g., ${}^{b}\backslash_{c}\langle !v_q\rangle(?c_q).Q$, the synchronization ${}^{a}\backslash_{b}^{b}\backslash_{c}$ has a final cost $v = v_p + v_q$ and it can actually happen if v satisfies both c_p and c_q. As we shall see, the *control* mechanism due to constraints have interesting properties and, in the quest of defining such extensions, we found solutions to distributed problems that do not have a simple solution in other process calculi. Quoting José Meseguer [16], "*increased expressiveness is not a theoretical luxury, but an eminently practical goal, since formal specification languages should describe as simply and naturally as possible the widest possible class of systems.*". This is precisely our goal here.

Contributions and organization. We start in Section 2 recalling the CNA framework. In Section 3 we introduce the notion of constraints based on complete lattices and monoids. Such structure allows us to compare values (e.g, $c < 42$) and also to accumulate values (e.g., adding the length/cost of two paths in a graph). We then extend the language of CNA to allow processes to communicate such values and also to query constraints on them. Hence, a transition can happen only if all the involved agents satisfy their own constraints. We call this new calculus constrained CNA (CCNA). We shall show that CCNA is a conservative extension of CNA (Notation 3.7): CNA processes can be seen as CCNA processes that exchange the less restrictive value and query the always-true constraint. In Section 3.2 we endow CCNA with a suitable operational semantics. This semantics has some novelties wrt the standard one for CNA: we follow a *late* approach in contrast to the *early* one used in [5]. Our semantics is then a good middle point between the inherent non-deterministic early semantics and the (more involved) symbolic semantics in [8]. Section 3.3 introduces the notion of (network) bisimulation for CCNA process and we prove that this equivalence is a congruence (thus allowing us to replace equals by equals in larger systems). In Section 4.1 we show that the use of constraints leads to a simple solution for the Dinning Philosopher problem where concurrent processes compete for the use of some resources. Our solution is deadlock-free and fair: all the agents can progress infinitely often. Fairness is usually imposed as an external condition but here, the constrained transition system satisfies such property. In Section 4.2 we model a graph representing a transportation system and we show how constraints allow us to discard some (undesired) behaviors. Moreover, in Section 4.3, we present an application in the context of a Service Level Agreement (SLA) protocol where constraints naturally represent restrictions such as the upper bound for the price of the service to be paid and the minimal quality (bandwidth) the client is expecting. Section 5 concludes the paper and present related work. We have also implemented a tool using the Maude system [16]. We shall not describe in depth such system here but we shall exemplify its use in Section 5. We thus contribute with a formal framework to specify constrained multiparty interactions and a prototypical tool showing its appropriateness as (automatic) reasoning technique.

2 Background on link chained interactions

Let $\mathcal{C}$ be the set of channel names, ranged over by $a, b, c, \ldots$, and $\tau, \Box$ two distinguished symbols not in $\mathcal{C}$. $\mathcal{A} = \mathcal{C} \cup \{\, \tau \,\} \cup \{\, \Box \,\}$ is the set of actions, ranged over by $\alpha, \beta, \ldots$, where the symbol τ denotes a *silent* action, while the symbol $\Box$ denotes a *virtual* (non-specified) action.

Definition 2.1 (Links: solid, virtual and valid) *A* link *is a pair $\ell = {}^{\alpha}\backslash_{\beta}$, with $\alpha,\ \beta \in \mathcal{A}$. We call α the* source site *of ℓ and β the* target site *of ℓ. A link ${}^{\alpha}\backslash_{\beta}$ is* solid *if $\alpha, \beta \neq \ \Box$; the link ${}^{\Box}\backslash_{\Box}$ is called* virtual*. A link is* valid *if it is solid or virtual. We shall use $\mathcal{L}$ to denote the set of valid links.*

Intuitively, ${}^{a}\backslash_{b}$ records an action a (or equivalently an input on channel a) and a co-action b (or an output on b), such that the input on a (receiving from the source of a communication) can be forwarded (to the target of the communication) along channel b. The τ-action is used when no interaction is required on the left (as in ${}^{\tau}\backslash_{a}$), on the right (as in ${}^{a}\backslash_{\tau}$) or on both sides (as in ${}^{\tau}\backslash_{\tau}$). The link ${}^{\tau}\backslash_{\tau}$ is called *τ-link*. A virtual link ${}^{\Box}\backslash_{\Box}$ represents a non-specified interaction that will be later completed. Examples of valid links are ${}^{\Box}\backslash_{\Box}$, ${}^{a}\backslash_{a}$, ${}^{\tau}\backslash_{a}$, ${}^{b}\backslash_{a}$, ${}^{\tau}\backslash_{\tau}$. The links ${}^{\Box}\backslash_{a}$, ${}^{a}\backslash_{\Box}$ are not valid.
Links can be combined in *link chains*. Intuitively, a link chain $s = \ell_1 \ldots \ell_n$ represents a multiparty interaction where each ℓ_i records the source and the target sites of each hop.

Definition 2.2 (Link Chain) *A* link chain *is a finite sequence $s = \ell_1 \ldots \ell_n$ of valid links $\ell_i = {}^{\alpha_i}\backslash_{\beta_i}$. We say that s is valid if:*

***(i)** for all $i \in [1, n)$, $\begin{cases} \beta_i, \alpha_{i+1} \in \mathcal{C} & \textit{implies } \beta_i = \alpha_{i+1} \\ \beta_i = \tau & \textit{iff } \alpha_{i+1} = \tau \end{cases}$; and*

***(ii)** $\exists i \in [1, n].\ \ell_i \neq {}^{\Box}\backslash_{\Box}$.*

We shall use VC to denote the set of valid link chains. The length *of a link chain s (i.e., the number of links in s) will be denoted as $|s|$.*

Condition (i) says that two adjacent solid links must match on their adjacent sites. In order to highlight such a matching, we shall write link chains as, e.g., ${}^{a}\backslash^{b}_{b}\backslash^{c}_{c}\backslash_{d}$ instead of sequences of links as in ${}^{a}\backslash_{b}\ {}^{b}\backslash_{c}\ {}^{c}\backslash_{d}$. Condition (i) also says that the silent action τ cannot be matched by a virtual action $\Box$. As we shall see, this is required since a τ-action can be only matched with τ when processes synchronize on restricted channels. Condition (ii) says that a valid link chain must have at least one solid link (i.e., the chain ${}^{\Box}\backslash^{\Box}_{\Box}\backslash_{\Box}$ is not valid). Some examples of valid link chains are: ${}^{\Box}\backslash^{a}_{\Box}\backslash^{b}_{b}\backslash_{\tau}$, ${}^{a}\backslash^{\Box}_{b}\backslash^{c}_{\Box}\backslash_{d}$, and ${}^{\tau}\backslash^{a}_{a}\backslash_{\tau}$. The first chain represents an interaction where there is a pending synchronization on the left of ${}^{a}\backslash_{b}$; similarly, the second chain represents an interaction where a third-party process must offer a link joining b and c (i.e., ${}^{b}\backslash_{c}$). Finally, the last chain is the result of a binary interaction between a process performing the output ${}^{\tau}\backslash_{a}$ and a process performing the input ${}^{a}\backslash_{\tau}$. The following are examples of link chains that are not valid: ${}^{a}\backslash^{c}_{b}\backslash_{d}$, ${}^{\Box}\backslash^{\tau}_{\Box}\backslash_{a}$, and ${}^{a}\backslash^{c}_{\tau}\backslash_{d}$. Hereafter, we only consider valid links and valid link chains. We shall use $\perp$ to denote non-valid links and chains.

Now we introduce the merge operator $s \bullet s'$ that acts on two link chains of the same length, i.e. $|s| = |s'|$. Intuitively, $s \bullet s'$ makes the two link chains collapse into one link chain where some of the virtual links in s (resp. s') have been substituted with the corresponding solid links in s' (resp. s).

Definition 2.3 (Merge) *Let $\alpha, \beta \in \mathcal{A}$ be actions. The merge operator on actions is defined as follow:*

$$\alpha \bullet \beta = \alpha \ \textit{if}\ \beta = \Box \qquad\qquad \alpha \bullet \beta = \beta \ \textit{if}\ \alpha = \Box \qquad\qquad \alpha \bullet \beta = \perp \ \textit{otherwise}$$

For links, let $\ell_1 = {}^{\alpha_1}\backslash_{\beta_1}$ and $\ell_2 = {}^{\alpha_2}\backslash_{\beta_2}$ be valid links and $\alpha_1 \bullet \alpha_2 = x_\alpha$, $\beta_1 \bullet \beta_2 = x_\beta$. If $x_\alpha, x_\beta \neq \perp$, then $\ell_1 \bullet \ell_2 = {}^{x_\alpha}\backslash_{x_\beta}$. Otherwise, $\ell_1 \bullet \ell_2 = \perp$. For the merge on chains, let $s = \ell_1 \ldots \ell_n$ and $s' = \ell'_1 \ldots \ell'_n$ be valid chains with $\ell_i = {}^{\alpha_i}\backslash_{\beta_i}$ and $\ell'_i = {}^{\alpha'_i}\backslash_{\beta'_i}$. If $\ell_i \bullet \ell'_i \neq \perp$ for all $i \in [1, n]$ and $(\ell_1 \bullet \ell'_1) \ldots (\ell_n \bullet \ell'_n)$ is a valid chain, then

$s \bullet s' = (\ell_1 \bullet \ell'_1)...(\ell_n \bullet \ell'_n)$. *Otherwise,* $s \bullet s' = \bot$.

As an example, the chains ${}^{\square}\backslash^{\square}_{\square}\backslash^{a}_{\square}\backslash_b$ and ${}^{c}\backslash^{\square}_{a}\backslash_{\square}$ cannot merge, as they have different length; ${}^{a}\backslash^{\square}_{b}\backslash_{\square}$ and ${}^{\square}\backslash^{c}_{\square}\backslash_d$ cannot merge since ${}^{a}\backslash^{c}_{b}\backslash_d$ is not a valid chain; a chain s cannot merge with itself; finally, ${}^{c}\backslash^{\square}_{a}\backslash^{b}_{\square}\backslash_d$ and ${}^{\square}\backslash^{a}_{\square}\backslash^{\square}_{b}\backslash_{\square}$ merge into ${}^{c}\backslash^{a}_{a}\backslash^{b}_{b}\backslash_d$.

As usual in process calculi, names are restricted in order to force interactions. Let $s = \ell_1...\ell_n = \ldots {}^{\alpha_i}\backslash^{\alpha_{i+1}}_{\beta_i}\backslash_{\beta_{i+1}} \ldots$ be a link chain. We say that a is *matched* in s if both

(1) $a \neq \alpha_1$ and $a \neq \beta_n$ (i.e., a cannot occur in the endpoints), and
(2) for any $i \in [1, n)$, either $\beta_i = \alpha_{i+1} = a$ or $\beta_i, \alpha_{i+1} \neq a$.

Otherwise, we say that a is *unmatched* (or *pending*) in s. For example, a and b are matched in ${}^{\tau}\backslash^{a}_{a}\backslash^{b}_{b}\backslash_d$. Instead, neither a nor b are matched in ${}^{d}\backslash^{\square}_{a}\backslash^{c}_{\square}\backslash_b$.

Definition 2.4 (Restriction) *Let $\alpha, \beta \in \mathcal{A}$ be actions and $a \in \mathcal{C}$ be a channel name. We define the following operations on actions and links:*

$$(\nu\, a)\alpha = \begin{cases} \tau & \text{if } \alpha = a \\ \alpha & \text{otherwise} \end{cases} \qquad \text{and} \qquad (\nu\, a)\ {}^{\alpha}\backslash_{\beta} = {}^{((\nu\, a)\alpha)}\backslash_{((\nu\, a)\beta)}$$

We lift those operations to link chains as follow:

$$(\nu\, a)s = \begin{cases} ((\nu\, a)\ell_1)\ldots((\nu\, a)\ell_n) & \text{if } a \text{ is matched in } s \\ \bot & \text{otherwise} \end{cases}$$

For instance, in $s = {}^{\tau}\backslash^{a}_{a}\backslash^{\square}_{b}\backslash_{\square}$, the name a is matched and $(\nu a)s = {}^{\tau}\backslash^{\tau}_{\tau}\backslash^{\square}_{b}\backslash_{\square}$; whereas $(\nu b)s$ is undefined since b is pending in s.

3 CNA with constraints

In this section we present the CNA calculus equipped with data passing and constrains on those data. Unlike the link-calculus [5] (and also the π-calculus in that respect), we will consider values which are not channel names, thus they will have a separate definition set. Our goal is to perform some checks on the data that participants in a interaction can share. Values will be taken from an algebraic structure that allows us to compare ($\preceq$) and also to accumulate ($\otimes$) those values. The following definitions are from [11] that simplifies at some extent the presentation of c-semiring [3] (another algebraic structure used for accumulating and comparing constraints).

Recall that a partial order is a pair $\langle \mathcal{D}, \preceq \rangle$ such that $\mathcal{D}$ is a set and $\preceq$ is a reflexive, transitive and antisymmetric relation on $\mathcal{D}$. We use $\prec$ to denote the strict version of $\preceq$ (i.e., $v \prec v'$ if $v \preceq v'$ and $v \neq v'$). We also use $\succeq$ and $\succ$ with the expected meaning. A complete lattice is a partial order where any subset $X \subseteq \mathcal{D}$ has a *least upper bound* denoted as $\sqcup X$. By duality, the *greatest lower bound*, denoted as $\sqcap X$, also exists. As usual, we use $\top$ and $\bot$ to denote, respectively, $\sqcup \mathcal{D}$ and $\sqcup \emptyset$. An abelian (or commutative) monoid is a triple $\langle \mathcal{D}, \otimes, \top \rangle$ where $\otimes : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is a commutative and associative operator and $\top$ is its identity (i.e., $\forall v \in \mathcal{D}. v \otimes \top = \top \otimes v = v$).

Definition 3.1 (CLM [11]) *A complete lattice monoid (for short CLM) is a tuple $\mathcal{K} = \langle \mathcal{D}, \preceq, \otimes \rangle$ s.t. $\langle \mathcal{D}, \preceq \rangle$ is a complete lattice, $\bot$ and $\top$ are, respectively, the least and the greatest elements of $\mathcal{D}$ and $\langle \mathcal{D}, \otimes, \top \rangle$ is a commutative monoid. We assume that $\otimes$ distributes over glb's, i.e.,*

$$\forall v \in \mathcal{D}. \forall X \subseteq \mathcal{D}. v \otimes \sqcap X = \sqcap \{ v \otimes x \mid x \in X \}$$

Let us explain the above definition with a concrete instance that we shall use in the following sections. Consider the structure $\mathcal{K}_{\mathbb{N}} = \langle \mathbb{N}^{\infty}_{+}, \geq, + \rangle$ where $\mathbb{N}^{\infty}_{+}$ is the set of natural numbers completed with ∞ and $\geq$ is the usual "greater than" relation on $\mathbb{N}^{\infty}_{+}$ (e.g., $5 \geq 2$). We can think of the elements in $\mathbb{N}^{\infty}_{+}$ as costs. Note that we consider 0 as the "best" ($\top$) cost ($x \preceq 0$, i.e., $x \geq 0$ for any $x \in \mathbb{N}^{\infty}_{+}$). Also, ∞ is the worst ($\bot$) cost we can assume. We accumulate costs with $+$ (addition on $\mathbb{N}$). When costs are added, we get "worse costs" (e.g., $3 + 2 \geq 3$). More generally, we can show that $\otimes$ is an *intensive operator*: $\forall v, v' \in \mathcal{D}, v \otimes v' \preceq v$. It is also possible to prove that $\bot$ is absorbing for $\otimes$ (i.e., $\forall v \in \mathcal{D}. v \otimes \bot = \bot$). In our concrete example, $v + \infty = \infty$.

Definition 3.2 (Residuation) *Let $\mathcal{K} = \langle \mathcal{D}, \preceq, \otimes \rangle$ be a CLM and $v, v' \in \mathcal{D}$. The residuation of v with respect to v' is defined as $v \div v' = \sqcap \{ x \in \mathcal{D} \mid v \preceq v' \otimes x \}$.*

As expected, $v \preceq v' \otimes (v \div v')$ (due to distributivity). Moreover, if $v' \preceq v$ then $v \div v' = \top$. In $\mathcal{K}_{\mathbb{N}}$, $\div$ is subtraction on $\mathbb{N}^{\infty}_{+}$ ($v \div v' = v - v'$) if $v \geq v'$ (i.e., $v \preceq v'$) and 0 otherwise. (See [2] for a discussion on residuation in constrain-based formalisms).

It is possible to combine different CLMs in order to measure/compare different entities in a single operation.

Lemma 3.3 (Combining CLMs) *Let $\mathcal{K}_1 = \langle \mathcal{D}_1, \preceq_1, \otimes_1 \rangle$ and $\mathcal{K}_2 = \langle \mathcal{D}_2, \preceq_2, \otimes_2 \rangle$ be CLMs. Define $\mathcal{K} = \mathcal{K}_1 \times \mathcal{K}_2 = \langle \mathcal{D}_1 \times \mathcal{D}_2, \prec, \otimes \rangle$ where $(v, w) \preceq (v', w')$ iff $v \preceq_1 v'$ and $w \preceq_2 w'$; and $(v, w) \otimes (v', w') = (v \otimes_1 v', w \otimes_2 w')$. Then, $\mathcal{K}$ is a CLM.*

Proof. The existence of arbitrary lubs is an easy consequence of the existence of lubs in $\mathcal{K}_1$ and $\mathcal{K}_2$. Distributivity also follows easily. Note that we can also define (point-wise) residuation on $\mathcal{K}$. □

3.1 Constrained multiparty interactions

Now we are ready to introduce the syntax of constrained CNA, from now on denoted as CCNA, which is parametric with respect to a CLM. In the following, we fix the CLM $\mathcal{K} = \langle \mathcal{D}, \preceq, \otimes \rangle$ with residuation operator $\div$. We shall use u, v to range over elements of $\mathcal{D}$. We shall call *data-variables* to variables (usually denoted as $x, y, ...$) that take values from $\mathcal{D}$.

Definition 3.4 (Syntax of CCNA) *Given a set of channel names $\mathcal{C}$ (ranged over by a, b) and a CLM $\mathcal{K}$, processes in CCNA are built from the syntax*

$$\begin{array}{llll}
exp & ::= \mathfrak{a} \;\mid\; v \;\mid\; x \;\mid\; exp \otimes exp \;\mid\; exp \div exp & & \textit{expressions} \\
atm & ::= exp \star exp' \quad \text{where } \star \in \{\preceq, \prec, =, \neq, \succeq, \succ\} & & \textit{atomic constraints} \\
c, c' & ::= atm \;\mid\; c \wedge c' & & \textit{constraints} \\
p, q & ::= \mathbf{0} \;\mid\; \ell\langle !e \rangle(?c).P \;\mid\; p + q & & \textit{sequential processes} \\
P, Q & ::= p \;\mid\; P \mid Q \;\mid\; (\nu\, a)P \;\mid\; A\langle a_1, \ldots, a_n; e_1, \ldots, e_m \rangle & & \textit{processes}
\end{array}$$

where $\mathfrak{a}$ is a distinguished data-variable representing the accumulated value of an interaction; $v \in \mathcal{D}$; x is a data-variable; e is an expression where $\mathfrak{a}$ does not occur; $\ell = {}^{\alpha}\backslash_{\beta}$ is a solid link built from $\mathcal{C} \cup \{\tau\}$ (i.e. $\alpha, \beta \neq \Box$); A is a process identifier for which we assume a (possibly recursive) definition of the form $A(\boldsymbol{b}; \boldsymbol{x}) \triangleq P$ where $\boldsymbol{b}$ is a list of pair-wise distinct names and $\boldsymbol{x}$ is a list of pair-wise distinct data-variables; and e_i is an expression where $\mathfrak{a}$ does not occur. □

In the following paragraphs, we explain in detail each of the components of this definitions. The symbol $\mathfrak{a}$ denotes a special data-variable that accumulates (using $\otimes$) the values added by each of the participants in an interaction (Definition 3.10 below). In $\mathcal{K}_{\mathbb{N}}$, examples of valid expressions are 3, $5 - y$, $5 + x + \mathfrak{a}$, etc.

Constraints will be used to check if the agents *agree* on the values of a given interaction. In $\mathcal{K}_{\mathbb{N}}$, examples of atomic constraints are $5 \geq 3$, $x \geq 4$, $x \geq y + 3$, $\mathfrak{a} \geq 10$, etc. A constraint is just a conjunction of such atoms. We use $\mathtt{tt}$ to denote the (always true) atomic constraint $\bot \preceq \top$.

Before explaining the meaning of processes, we need an extra definition. The only binder for (data-) variables is the process definition $A(\boldsymbol{b}; \boldsymbol{x}) \triangleq P$ where the variables in $\boldsymbol{x}$ occur bound in P. We shall use $fv(P)$ to denote the set of free (data-) variables in the process P. We shall also use $fv(c)$ to denote the set of free (data-) variables in c (including $\mathfrak{a}$).

Definition 3.5 (Entailment) *We say that a constraint c is ground if it does not contain data-variables nor the symbol $\mathfrak{a}$ (i.e., $fv(c) = \emptyset$). Given a ground atomic constraint $c = e \star e'$, we say that c holds, notation $\mathcal{K} \models c$, if e and e' reduce (performing the operations $\otimes$ and $\div$ in $\mathcal{K}$), respectively, to v and v' and the relation $v \star v'$ holds in $\mathcal{K}$. Otherwise, we write $\mathcal{K} \not\models c$. We extend this notion to ground constraints as follows: $\mathcal{K} \models c_1 \wedge \cdots \wedge c_n$ whenever $\mathcal{K} \models c_i$ for all $i \in 1..n$. Let c, c' be constraints s.t. $fv(c), fv(c') \subseteq \{\mathfrak{a}\}$. We say that c, c' are equivalent, notation $c \approx c'$ if $\forall v \in \mathcal{D}. \mathcal{K} \models c[v/\mathfrak{a}]$ iff $\mathcal{K} \models c'[v/\mathfrak{a}]$.*

As an example, we have the following entailments: $\mathcal{K}_{\mathbb{N}} \models 3 + 2 \preceq 1 + 2$; and $\mathcal{K}_{\mathbb{N}} \models 3 \div 2 \succeq 1 + 2$. Moreover, if $c_1 = \mathtt{tt} \wedge 3 \preceq \mathfrak{a}$ and $c_2 = \mathfrak{a} + 0 \succeq 5 \div 2$ then $c_1 \approx c_2$.

The process $\mathbf{0}$ does nothing. The process $\ell\langle !e \rangle(?c).P$ offers the *solid link* ℓ along with the value denoted by the expression e and *checks* whether the constraint c holds. After this interaction, the process behaves as P. The non-deterministic process $p + q$ can either behave as p or q. Interleaved parallel composition is denoted as $P \mid Q$. The process $(\nu\, a)P$ behaves as P but it cannot exhibit any action a. Hence, we can say that a is local (or private) in P. As usual, $(\nu\, a)P$ binds the free occurrences of a in P.

The call $A\langle a_1, ..., a_n; e_1, ..., e_m \rangle$ behaves as the process $P[a_1/b_1, ..., a_n/b_n][e_1/x_1, \ldots, e_m/x_m]$ if the constant A is defined as $A(b_1, ..., b_n; x_1, ..., x_m) \triangleq P$. As expected, the actual parameters substitute the formal parameters of the definition. Besides binding the variables x_i, the above definition binds the names b_i. We shall use $fn(P)$ and $bn(P)$ to denote, respectively, the free and bound names in P.

We shall impose some restrictions on process that are commonplace in the literature (see e.g., [12]).

Definition 3.6 (Valid Processes) *Processes are taken up to alpha-conversion (renaming of bound names). We assume that in a process definition $A(\boldsymbol{b};\boldsymbol{x}) \triangleq P$, $fn(P) \subseteq \boldsymbol{b}$ and $fv(P) \subseteq \boldsymbol{x}$. Moreover, in order to guarantee the transition system to be finitely branching, we assume that recursive calls in P must be guarded inside a prefix $\ell\langle !e\rangle(?c).Q$. Finally, we assume that in all processes, the occurrence of a data-variable x (different from $\mathfrak{a}$) is always bound by a process definition.*

For instance, the processes $\ell\langle !2\rangle(?x > 5).Q$ and $\ell\langle !2 + x\rangle(?\mathfrak{a} > 5).Q$ are not valid since x occurs free (and not bound by a process definition). Moreover, the process definition $A() \stackrel{\mathsf{def}}{=} A\langle\rangle|P$ is not valid since the call $A\langle\rangle$ is not guarded (thus making the transition system infinitely branching).

Notation 3.7 *For the sake of readability, we shall use the following shortcuts. We shall omit a trailing* $\mathbf{0}$*, e.g. by writing ${}^{a}\backslash_{b}$ instead of ${}^{a}\backslash_{b}.\mathbf{0}$. We shall also write $\ell\langle !e\rangle$ instead of $\ell\langle !e\rangle(?\,tt)$. Recall that $v \otimes \top = v$ for all $v \in \mathcal{D}$. Hence, we shall write $\ell(?c)$ instead of $\ell\langle !\top\rangle(?c)$. Finally, we shall simply write ℓ instead of $\ell\langle !\top\rangle(?\,tt)$. Note that any* CNA *process is also a* CCNA *process whose shared value ($\top$) is irrelevant for the final interaction and whose constraint always holds (tt).*

The following example shows how constraints can be used to limit the number of participants in a multiparty interaction. For the moment, the discussion about the behavior of process remains informal until we define the semantics in the next section.

Example 3.8 *The process $P = {}^{a}\backslash_{b}$ is able to interact with any other process, say $Q = {}^{c}\backslash_{d}$ and synchronize producing the link chain ${}^{a}\backslash^{\square}_{b}\backslash^{c}_{\square}\backslash_{d}$. Other outputs can be also expected from that interaction, as e.g. ${}^{c}\backslash^{\square}_{d}\backslash^{a}_{\square}\backslash_{b}$. In fact, a 3-party synchronization is also possible with yet another process $R = {}^{b}\backslash_{c}$, thus building the chain ${}^{a}\backslash^{b}_{b}\backslash^{c}_{c}\backslash_{d}$. More generally, P can interact with an unbounded number of other processes by building suitable (valid) link chains. This means that interactions in* CNA *are open since the number of participants is not fixed a priori. This is a quite expressive feature of the calculus but it makes also difficult to reason about processes. (See [7] and [8] for symbolic techniques to deal with this problem). Consider the structure $\mathcal{K}_{\mathbb{N}}$ and the processes below*

$$P' = {}^{a}\backslash_{b}\langle !1\rangle(?\mathfrak{a} \leq 2) \qquad Q' = {}^{c}\backslash_{d}\langle !1\rangle(?\mathfrak{a} \leq 2) \qquad R' = {}^{b}\backslash_{c}\langle !1\rangle(?\mathfrak{a} \leq 2)$$

Hence, P' can interact with at most one of the other two processes (Q' or R') since, in each interaction, the value 1 is accumulated and such value must be less than 2. This is a very intuitive (and declarative) mechanism for counting and restricting the participants in an interaction. □

Derived construct (tuples). Let us introduce an idiomatic construct that will be useful. Due to Lemma 3.3, we can assume that agents offer links with tuples of elements as, e.g., $(e_1, \ldots, e_n)$. Such tuples are elements of the CLM $\mathcal{K}_1 \times \cdots \times \mathcal{K}_n$. Hence, we shall use the more convenient notation

$$\ell\langle !e_1, \ldots, e_n\rangle(?c_1 \wedge \cdots \wedge c_n)$$

where each constraint c_i may use the special symbol $\mathfrak{a}_i$ (denoting the accumulated values in the i-th position of the tuple). Note that these tuples and constraints can be easily rewritten in the syntax of Definition 3.4 by using the construction in Lemma 3.3. Since each CLM $\mathcal{K}_i$ represents a different value/measure to be accumulated, it is not legal to combine values of different CLMs in the same expressions. For instance, expressions as, e.g., $\mathfrak{a}_1 \div \mathfrak{a}_2$ are not legal. Similarly for constraints. Next notational convention allows us to give alias for the "accumulating" variables $\mathfrak{a}_i$ to make specifications cleaner.

Notation 3.9 (Tuples and variables) *For a given specification, we can determine that tuples used in interactions are of the form $\langle x_1, ..., x_n\rangle$ where the aliases x_i cannot be bound by a process definitions. Hence, we write $\ell\langle !x_1 = e_1, \ldots, x_n = e_n\rangle(?c_1 \wedge \cdots \wedge c_n)$ to mean $\ell\langle !e_1, \ldots, e_n\rangle(?\,(c_1 \wedge \cdots \wedge c_n)\,[\mathfrak{a}_1/x_1, ..., \mathfrak{a}_n/x_n])$. Moreover, if $v_i = \top$, we further omit "$x_i = \top$". For instance, if we determine that tuples are of the form $\langle cost, speed\rangle$, the expression $\ell\langle !speed = 1\rangle(?cost \leq 3 \wedge speed \geq 10)$ means $\ell\langle !\top, 1\rangle(?\mathfrak{a}_1 \leq 3 \wedge \mathfrak{a}_2 \geq 10)$.*

3.2 Semantics

In this section we introduce the operational semantics for CCNA. A novelty in this semantics is the use of a *late* approach where the number of participants is inferred only in the communication rule in contrast to the *early* approach adopted in [4] (where the rule *Act* needs to "guess" the size of the interaction) . This distinction will be clarified in brief.

Before defining the semantics, we shall lift the definition of merge on link chains (Definition 2.3) to consider also *valued-constrained-chains* (VCC) of the form $s\langle !e\rangle(?c)$ where s is a link chain, e is a ground expression (no free variables) and c is a constraint where $fv(c) \subseteq \{\mathfrak{a}\}$. For that, we allow link-chains to be enlarged (injecting virtual links) or contracted via the relation $\bowtie$ defined below.

$$\frac{}{\ell\langle !e\rangle(?c).P \xrightarrow{\ell\langle !e\rangle(?c)} P}\ Act \qquad \frac{p \xrightarrow{\mu} P'}{p+q \xrightarrow{\mu} P'}\ Lsum \qquad \frac{q \xrightarrow{\mu} Q'}{p+q \xrightarrow{\mu} Q'}\ Rsum$$

$$\frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q}\ Lpar \qquad \frac{Q \xrightarrow{\mu} Q'}{P \mid Q \xrightarrow{\mu} P \mid Q'}\ Rpar \qquad \frac{P \xrightarrow{\mu} P' \quad Q \xrightarrow{\phi} Q'}{P \mid Q \xrightarrow{\mu\bullet\phi} P' \mid Q'}\ Com$$

$$\frac{P[\boldsymbol{a}/\boldsymbol{b}][\boldsymbol{v}/\boldsymbol{x}] \xrightarrow{\mu} P' \qquad A(\boldsymbol{b};\boldsymbol{x}) \triangleq P}{A\langle \boldsymbol{a};\boldsymbol{v}\rangle \xrightarrow{\mu} P'}\ Ide \qquad \frac{P \xrightarrow{\mu} P'}{(\nu a)P \xrightarrow{(\nu a)\mu} (\nu a)P'}\ Res$$

Fig. 1. Semantics. All the rules have the proviso that the labels in the transitions are valid (Def. 3.10).

Definition 3.10 (Valued-constrained-chains and operations) *We let $\bowtie$ be the least equivalence relation over link chains closed under the axioms (whenever both sides are valid link chains):*

$$s\ {}^{\square}\backslash_{\square} \bowtie s \qquad s_1\ {}^{\square}\backslash^{\square}_{\square}\backslash_{\square}\ s_2 \bowtie s_1\ {}^{\square}\backslash_{\square}\ s_2 \qquad {}^{\square}\backslash_{\square}\ s \bowtie s \qquad s_1\ {}^{\alpha}\backslash^{a}_{a}\backslash_{\beta}\ s_2 \bowtie s_1{}^{\alpha}\backslash^{\square}_{a}\backslash^{a}_{\square}\backslash_{\beta}\ s_2$$

We merge VCCs as follow: $s\langle !e\rangle(?c) \bullet s'\langle !e'\rangle(?c') = (w \bullet w')\langle !e \otimes e'\rangle(?c \wedge c')$ with $s \bowtie w$ and $s' \bowtie w'$. We define $(\nu a)(s\langle !e\rangle(?c))$ as $((\nu a)s)\langle !e\rangle(?c)$. We say that a VCC $s\langle !e\rangle(?c)$ is valid iff s is a valid link chain and $\mathcal{K} \models c[e/\mathfrak{a}]$.

We shall use μ, ϕ to range over VCCs. Note that the values are merged using the $\otimes$ operator of the CLM. Note also that, in order to check the validity of a VCC, the symbol $\mathfrak{a}$ is replaced with the current *accumulated* value e. Since we are assuming that there are no free data-variables in a process (see Definition 3.6), once the data-variables of a definition have been replaced with concrete values, $c[e/\mathfrak{a}]$ is indeed a ground constraint. Finally, since there are no occurrences of names in values and constraints, the restriction operation on VCCs acts only on the link chains (Definition 2.4) and hence, e and c remain the same.

The structural operational semantics is given by the labeled transition system $(\mathcal{P}, \mathit{VCC}, \longrightarrow)$ where the set $\mathcal{P}$ of states is the set of CCNA processes, labels are valid valued-constrained-chains (VCC) and the transition relation $\longrightarrow$ is the minimal transition relation generated by the rules in Figure 1.

The prefix process $\ell\langle !e\rangle(?c).P$ simply offers the link ℓ with value/expression e and checks whether c is valid, i.e., $\ell\langle !e\rangle(?c)$ must be a valid VCC (Rule *Act*). If p is able to exhibit a transition to P' with label μ, then $p+q \xrightarrow{\mu} P'$ (Rule *Lsum*). Similarly for q (Rule *Rsum*). If P can exhibit a transition, it can also exhibit the same transition when running in parallel with Q (Rules *Lpar* and *Rpar*). In *Res*, if P offers the action μ then $(\nu a)P$ offers $(\nu a)\mu$ (if it is valid). Rule *Ide* simply replaces the formal parameter with the actual parameters.

The synchronization mechanism (Rule *Com*) works by merging two VCCs. When doing that, note that the link chains can be enlarged (Definition 3.10) and hence, the links of one chain can be placed in an admissible position of the other chain. Note that the decision about the length of the resulting chain is postponed until the use of the rule *Com*. This is a different approach from the one considered in [4] (for CNA processes) where the size of the interaction must be inferred in the *Act* rule (by enlarging in this rule ℓ with $\bowtie$). We also note that contrary to CCS, the Rule *Com* in CCNA (and CNA) can appear several times in the proof tree of a transition since the merging operator can always inject more virtual links to allow other agents to participate as shown in the following example.

Example 3.11 *Consider the CLM $\mathcal{K}_{\mathbb{N}}$ and the processes:*

$$P = {}^{\tau}\backslash_{a}\langle !2\rangle(?\mathfrak{a} \leq 10).P' \qquad Q = {}^{a}\backslash_{b}\langle !3\rangle(?\mathfrak{a} \leq 12).Q' \qquad R = {}^{b}\backslash_{\tau}\langle !5\rangle(?\mathfrak{a} \geq 4).R'$$

representing three agents interested in building a house. Each of them has a cost for her services (e.g., P charges \$2). Moreover, P is not willing to participate in a project that costs more than \$10 and R does not participate in "small" projects with a cost below \$4. P requires someone providing a service matching its output link a. Q offers a and expects in exchange b and R provides b. The three agents can indeed engage in the project as the derivation in Figure 3.2 shows. Note that $(\nu a, b)^{\tau}\backslash^{a}_{a}\backslash^{b}_{b}\backslash_{\tau} = {}^{\tau}\backslash^{\tau}_{\tau}\backslash^{\tau}_{\tau}\backslash_{\tau}$. Also, in all the steps of this derivation the labels of the transitions are valid VCCs. Rule Com is used twice in the derivation, thus resulting in a 3-party interaction.

If we define P_2 as ${}^{\tau}\backslash_{a}\langle !2\rangle(?\mathfrak{a} < 10).P_2'$ the process $(\nu a, b)(P_2|Q|R)$ does not have any transition since $s\langle !10\rangle(?\mathfrak{a} < 10 \wedge \mathfrak{a} \leq 12 \wedge \mathfrak{a} \geq 4)$ is not a valid VCC. Moreover the names a, b are restricted and none of the processes in $P_2|Q|R$ can evolve independently using the rules Lpar and Rpar (e.g., a is not matched in ${}^{\tau}\backslash_{a}$ and

$$\dfrac{\dfrac{\dfrac{}{P \xrightarrow{{}^{\tau}\backslash_{a}\langle !2\rangle(?\mathfrak{a}\leq 10)} P'}\;Act \qquad \dfrac{\dfrac{}{Q \xrightarrow{{}^{a}\backslash_{b}\langle !3\rangle(?\mathfrak{a}\leq 12)} Q'}\;Act \quad \dfrac{}{R \xrightarrow{{}^{b}\backslash_{\tau}\langle !5\rangle(?\mathfrak{a}\geq 4)} R'}\;Act}{Q|R \xrightarrow{{}^{a}\backslash^{b}_{b}\backslash_{\tau}\langle !8\rangle(?\mathfrak{a}\leq 12\wedge\mathfrak{a}\geq 4)} Q'|R'}\;Com}{P|Q|R \xrightarrow{{}^{\tau}\backslash^{a}_{a}\backslash^{b}_{b}\backslash_{\tau}\langle !10\rangle(?c)} P'|Q'|R'}\;Com}{(\nu a,b)(P|Q|R) \xrightarrow{{}^{\tau}\backslash^{\tau}_{\tau}\backslash^{\tau}_{\tau}\backslash_{\tau}\langle !10\rangle(?\mathfrak{a}\leq 10\wedge\mathfrak{a}\leq 12\wedge\mathfrak{a}\geq 4)} (\nu a,b)(P'|Q'|R')}\;2\times Res$$

Fig. 2. Derivation in Example 3.11.

then, $(\nu a)^{\tau}\backslash_{a}$ is not valid). In words, P_2 refuses a 3-party interaction with Q and R since the project will cost more than she expects. □

Example 3.12 (Conditionals) *Given an atomic constraint $c = e_1 \star e_2$, let $\widehat{c}$ denote the atomic constraint $e_1 \widehat{\star} e_2$ where $\widehat{\star}$ substitutes $=$ with $\neq$, $\preceq$ with $\succ$, $\prec$ with $\succeq$, etc. It is straightforward to see that: (1) $\widehat{\widehat{c}} = c$; and (2) given a ground atomic constraint c, $\mathcal{K} \models c$ iff $\mathcal{K} \not\models \widehat{c}$. We can define a conditional construct to select the continuation of a process depending on the entailment of a constraint. More precisely, if $c = c_1 \wedge \cdots \wedge c_n$ where each c_i is an atomic constraint, we can write $\ell\langle !e\rangle(\texttt{if}\ ?c\ \texttt{then}\ P\ \texttt{else}\ Q)$ to denote the process $\ell\langle !e\rangle(?c).P + \ell\langle !e\rangle(?\widehat{c_1}).Q + \cdots + \ell\langle !e\rangle(?\widehat{c_n}).Q$. Note that P is executed whenever all c_i hold (and hence c holds) and Q is executed if there is a c_i that does not hold in the underlying CLM.* □

Definition 3.13 (Computations) *Let W^* be the set of finite and infinite sequences of valid VCCs. Let P be a process and $\sigma = s_1.s_2... \in W^*$ be an infinite sequence. We say that σ is a computation of P if $P = P_0 \xrightarrow{s_1} P_1 \xrightarrow{s_2} P_2 \xrightarrow{s_3} \cdots$. If P cannot afford any transition, we shall write $P \not\longrightarrow$ and we call P a* dead-lock. *If $\sigma = s_1.s_2...s_n \in W^*$ is a finite sequence, we say that σ is a (finite) computation of P if $P = P_0 \xrightarrow{s_1} P_1 \cdots P_{n-1} \xrightarrow{s_n} P_n \not\longrightarrow$. In both cases we shall write $P \xrightarrow{\sigma}$. We shall use $\mathcal{O}(P) \subseteq W^*$ to denote the set $\{\sigma \mid P \xrightarrow{\sigma}\}$.*

3.3 Network bisimulation

Now we define a behavioral equivalence on CCNA processes that we show to be a congruence. In the tradition of CNA, we do not distinguish between processes that exhibit different internal transitions. This is reflected in the following extension of the equivalence relation $\bowtie$ originally proposed in [4].

Definition 3.14 (Equivalence $\bowtie$) *We let $\bowtie$ be the least equivalence relation over link chains closed under the following inference rules:*

$$\frac{s \blacktriangleright\!\!\blacktriangleleft s'}{s \bowtie s'} \qquad\qquad s_1\, {}^{\alpha}\backslash^{\tau}_{\tau}\backslash_{\beta}\, s_2 \bowtie s_1\, {}^{\alpha}\backslash_{\beta}\, s_2$$

We lift such relation to VCC as follows: $s\langle !e\rangle(?c) \bowtie s'\langle !e\rangle(?c')$ iff $s \bowtie s'$ and $c \approx c'$.

Definition 3.15 (Network Bisimulation) *A* network bisimulation $\mathbf{R}$ *is a binary relation over* CCNA *processes such that, if $P\ \mathbf{R}\ Q$ then:*

- *if $P \xrightarrow{\mu} P'$, then $\exists\ \phi$, Q' such that $\mu \bowtie \phi$, $Q \xrightarrow{\phi} Q'$, and $P'\ \mathbf{R}\ Q'$;*
- *if $Q \xrightarrow{\mu} Q'$, then $\exists\ \phi$, P' such that $\mu \bowtie \phi$, $P \xrightarrow{\phi} P'$, and $P'\ \mathbf{R}\ Q'$.*

P and Q are network bisimilar, *notation $P \sim Q$, if there exists a network bisimulation $\mathbf{R}$ s.t. $P\ \mathbf{R}\ Q$.*

Following standard techniques (see e.g., [12,23]), we can show that $\sim$ is an equivalence relation and it is the largest network bisimulation relation.

Let us give some illustrative examples with the structure $\mathcal{K}_{\mathbb{N}}$. For any P, $\mathbf{0} \sim \ell\langle !3\rangle(?\mathfrak{a} \leq 2).P$ since $\mathcal{K}_{\mathbb{N}} \not\models 3 \leq 2$. Moreover, merging constraints cannot make $\mathfrak{a} \leq 2$ valid (due to intensiveness of $\otimes$ w.r.t. $\preceq$, i.e., $\mathfrak{a}$ will be always greater than 3).

The processes $P = \ell\langle !3\rangle(?\mathfrak{a} \leq 5)$ and $Q = \ell\langle !3\rangle(?\mathfrak{a} \leq 7)$ cannot be considered equivalent: Q can, for instance, synchronize with $R = \ell'\langle !4\rangle$ while P cannot. Now consider $P = \ell\langle !2\rangle(?\mathfrak{a} \leq 2)$ and $Q = \ell\langle !4\rangle(?\mathfrak{a} \leq 4)$. Note that both processes can synchronize with a process of the form $R = \ell'\langle !0\rangle(?c)$. However, if $c = \mathfrak{a} \leq 2$, P and R can synchronize but Q and R cannot.

Next we show that $\sim$ is a congruence relation and then, we can replace "equals by equals" in any context. For that, let $\mathcal{C}[\cdot]$ denote a process expression with a single occurrence of a hole $[\cdot]$. Moreover, if P is a process, $\mathcal{C}[P]$ denotes a process expression resulting from the substitution of the hole $[\cdot]$ with P.

Theorem 3.16 (Congruence) *If $P \sim Q$ then, for any context $\mathcal{C}[\cdot]$, $\mathcal{C}[P] \sim \mathcal{C}[Q]$.*

The above theorem is proved by exhibiting appropriate network bisimulations for any case/context. The complete proof is in the appendix.

4 Applications: fairness and constrained interactions

In this section we give three compelling examples showing how to declaratively control multiparty interactions by means of constraints. The first example is the canonical problem of the dinning philosophers. In this case, by adding constraints, we are able to specify a deadlock free and fair solution for the problem. The second example models a network transportation system where constraints may represent costs or temporal restrictions. In our last example, constraints are used to model service level agreements in a negotiation protocol.

4.1 The dinning philosophers

The classical example of the dining philosophers (DP) has been introduced to study interactions between concurrent entities that want to share some resources. The problem relates n philosophers sitting around a table, where each one has its own dish, and they can only eat or think. When they, independently, decide to eat, they need two forks. On the table, there is only one fork between two dishes, i.e., exactly n forks.

It is well known there is no symmetric and deadlock-free specification of this system using only binary interactions [12] as in, e.g., CCS. Let us illustrate the problem considering only two philosophers. The philosophers are specified as the CCS process $P_i = f_i.f_{(i+1)mod\ 2}.\overline{eat_i}.P_i'$ where P_0 first grabs the fork 0, then he grabs the fork 1 to later start eating (similarly for P_1). If we run in parallel P_0 and P_1 along with the processes specifying the two forks ($F_i = \overline{f_i}.F_i'$), the system can reach a deadlock when P_0 takes the fork 0 and P_1 takes the fork 1.

By using a multiparty synchronization calculus, the DP problem has a simple and very natural deadlock free specification (see [7,8] for a solution using CNA and [12] for a solution using Multi-CCS). In that case, in an atomic (or multiparty) interaction, a philosopher takes at the same time both forks, thus avoiding the deadlock situation described above. However, the solutions in Multi-CCS and CNA may exhibit unfair computations where, e.g., a given philosopher eats or thinks all the time (and the others cannot progress).

Definition 4.1 (Fairness [24]) *Let π be an infinite computation, $\pi = P_0 \xrightarrow{s_1} P_1 \xrightarrow{s_2} P_2 \xrightarrow{s_3} \cdots$. A VCC μ is relentlessly enabled in π if $\forall\ \pi''$, π' s.t. $\pi = \pi''\pi'$, π' contains a process P_i that can afford a transition labeled with μ. Moreover, π is strongly fair if each relentlessly enabled VCC μ on π' occurs in π'.*

In words, a computation π is strongly fair if an action (VCC) that is relentlessly enabled in π, occurs infinitely often in π.

Here we focus on a fair solution for the DP problem: due to deadlock-freeness, every computation is infinite and, by fairness, in every computation each philosopher eats infinitely many times. For that, we use constraints to neatly implement a sort of ticket service, thus guaranteeing that philosophers must alternate the use of the forks. From now on, we fix the CLM to be the structure $\mathcal{K}_{\mathbb{N}}$.

Below we describe our first attempt to solve the problem. Unfortunately, the specification is deadlock-free but it is not fair. We shall use $DP(n)$ to denote the instance of the DP problem with n philosophers and i_n^+ to denote $(i+1)mod\ n$.

Example 4.2 (Dinning Philosophers) *Let $n \geq 2$ be the number of philosophers (and forks) in the problem and define $Fork_i$ with $i \in [0,n)$ as follows:*

$$\begin{aligned} Fork_i(l,r) \triangleq\ & {}^{\tau}\backslash_{\tau}(?l = 0 \wedge r = 0).Fork_i\langle N,N\rangle \\ & + {}^{\tau}\backslash_{upL_i}(?l > 0).{}^{\tau}\backslash_{dw_i}.Fork_i\langle l\text{-}1, r\rangle \quad + \quad {}^{upR_i}\backslash_{\tau}(?r > 0).{}^{dw_i}\backslash_{\tau}.Fork_i\langle l, r\text{-}1\rangle \end{aligned}$$

In this specification, N is a global parameter/constant of the model indicating how many times we allow a philosopher to consecutively use the forks. The parameter l of the definition is the maximum number of times the philosopher on the left can use the i-th fork. Similarly, for the parameter r. Every time the i-th fork is used by the philosopher on its left (resp. right), the parameter l (resp. r) is decremented (using $\div$) by 1. The process $Fork_i(l,r)$ can reset its parameters to the initial values only when both l and r are equal to zero. The values of the parameters are checked by the constraints associated to the prefix ${}^{\tau}\backslash_{upL_i}$ (resp. ${}^{upR_i}\backslash_{\tau}$) that allow the philosopher on the left (resp. right) to grab the fork.

The specification of the philosophers is as follows:

$$Phil_i() \triangleq {}^{\tau}\backslash_{tk_i}.Phil_i\langle\rangle + {}^{upL_i}\backslash_{upR_{i_n^+}}.{}^{\tau}\backslash_{eat_i}.{}^{dw_i}\backslash_{dw_{i_n^+}}.Phil_i\langle\rangle$$

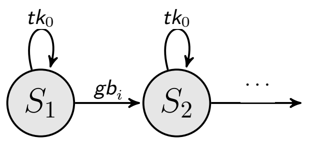
Hence, the process $Phil_i\langle\rangle$ can either think or establish a 3-party synchronization with the forks on his right and on his left. In that case, he can eat to later release both forks in another 3-party synchronization. In fact, *the release of the forks can be done independently and there is no need for a multiparty synchronization.*

The whole system restricts all the channel names but eat_i which is a visible action:

$$DP = (\nu\, \widetilde{upL_i}, \widetilde{upR_i}, \widetilde{dw_i})(Phil_0|\dots|Phil_{n-1}|Fork_0(N,N)|\dots|Fork_{n-1}(N,N))$$

Here $\widetilde{upL_i}, \widetilde{upR_i}, \widetilde{dw_i}$ stand for the sets of channel names used in $Phil_i$ and $Fork_i$ with $i \in [0, n\text{-}1]$. □

The transition system generated by the process DP (that can be computed with our tool, see Section 5) is indeed deadlock free. Moreover, once P_i has used the forks N times, he has to wait until his neighbors eat also N times to be able to eat again. This means that P_i cannot take control of the forks forever and, at some point, he has to wait for the others. In other words, there are no computations where, e.g., P_i eats infinitely many times and P_j can never grab the forks.



In this model, however, we cannot prove that P_i can eat infinitely many times. The problem is the *thinking* action: there is an infinite computation where, e.g., P_0 is always thinking and nobody is eating. Such computation corresponds to the loop on state S_1 in the abstract version of the transition system in the figure on the left. In this loop, the action gb_i, representing P_i grabbing the forks, is always enabled. This situation can be interpreted as "P_i has the potential of grabbing the forks but the scheduler never gives him the chance to do it".

The fair system can be obtained by controlling also the thinking action. Similar to the solution in [12], we can enforce that philosophers must eat after thinking, thus alternating between thinking and eating states. This is the purpose of moving to the state $Phil'_i$ after exhibiting the think action in the model below.

Example 4.3 (Fair alternating system) *Consider the processes definitions Fork and DP in Example 4.2 where the definition of $Phil_i$ is modified as follows:*

$Phil_i() \triangleq {}^{\tau}\backslash_{tk_i}.Phil'_i\langle\rangle + {}^{upL_i}\backslash_{upR_{i\underset{n}{+}}}.{}^{\tau}\backslash_{eat_i}.{}^{dw_i}\backslash_{dw_{i\underset{n}{+}}}.Phil_i\langle\rangle$

$Phil'_i() \triangleq {}^{upL_i}\backslash_{upR_{i\underset{n}{+}}}.{}^{\tau}\backslash_{eat_i}.{}^{dw_i}\backslash_{dw_{i\underset{n}{+}}}.Phil_i\langle\rangle$

For illustration, consider a possible transition where the philosopher n-1 takes both forks to later eat and release the forks:

$$DP \xrightarrow{(\nu\,\widetilde{n})^{\tau}\backslash^{upL_{n\text{-}1}}_{upL_{n\text{-}1}}\backslash^{upR_0}_{upR_0}\backslash_{\tau}(?1{>}0 \wedge ?1{>}0)} DP_1 \xrightarrow{{}^{\tau}\backslash_{eat_{n\text{-}1}}} DP_2 \xrightarrow{(\nu\,\widetilde{n})^{\tau}\backslash^{dw_{n\text{-}1}}_{dw_{n\text{-}1}}\backslash^{dw_0}_{dw_0}\backslash_{\tau}} DP_3$$

where $\widetilde{n} = \widetilde{upL_i}, \widetilde{upR_i}, \widetilde{dw_i}$; DP_1 is as DP but $Phil_0$ is replaced with ${}^{\tau}\backslash_{eat_i}.{}^{dw_i}\backslash_{dw_{i\underset{n}{+}}}.Phil_0\langle\rangle$; DP_2 is as DP but $Phil_0$ is replaced with ${}^{dw_i}\backslash_{dw_{i\underset{n}{+}}}.Phil_0\langle\rangle$; and, finally,

$DP_3 = (\nu\, \widetilde{upL_i}, \widetilde{upR_i}, \widetilde{dw_i})(Phil_0|\dots|Phil_{n-1}|Fork_0(N\text{-}1,N)|\dots|Fork_i(N,N)|\dots|Fork_{n\text{-}1}(N,N\text{-}1))$. *Note the new state of the forks 0 and n-1, namely, $Fork_0(N\text{-}1,N)$ and $Fork_{n\text{-}1}(N,N\text{-}1)$.* □

We can show that the process DP in the above example produces an alternating execution between the philosophers. For concreteness, consider only two philosophers and let DP_W be as $DP(2)$ where we ignore the *think* actions, i.e., we consider only the process $Phil'_i()$ calling to itself, instead of calling the process $Phil_i\langle\rangle$. We can define the following specification:

$$Spec() \stackrel{\text{def}}{=} {}^{\tau}\backslash_{eat_0}.{}^{\tau}\backslash_{eat_1}.Spec\langle\rangle \quad + \quad {}^{\tau}\backslash_{eat_1}.{}^{\tau}\backslash_{eat_0}.Spec\langle\rangle$$

stating that philosophers must alternate when $N = 1$. We can then prove the equivalence $D_W \sim Spec\langle\rangle$.

Let us now show that fairness holds even considering the *thinking* action. Without loss of generality, let $N = 1$. For readability, let us give a more abstract representation of the state of the forks as a ring of tuples of the form $\mathcal{S}_n = \langle a_0, a_1\rangle\langle a_1, a_2\rangle\langle a_2, a_3\rangle \dots \langle a_{n-1}, a_0\rangle$ where $a_i \in \{0, 1, ?\}$. Such ring is subject to the following transition rules:

- **Grab**: $\dots\langle x, 1\rangle\langle 1, y\rangle \dots \Rightarrow \dots\langle x, ?\rangle\langle ?, y\rangle\dots$ **Grab-0**: $\langle 1, x\rangle \dots \langle y, 1\rangle \Rightarrow \langle ?, x\rangle \dots \langle y, ?\rangle$
- **Grab-0-end**: $\langle ?, x\rangle \dots \langle y, ?\rangle \Rightarrow \langle 0, x\rangle \dots \langle y, 0\rangle$ **Grab-end**: $\dots\langle x, ?\rangle\langle ?, y\rangle \dots \Rightarrow \dots\langle x, 0\rangle\langle 0, y\rangle\dots$
- **Reset**: $\dots\langle 0, 0\rangle \dots \Rightarrow \dots\langle 1, 1\rangle \dots$.

The state of the fork i is represented by the i-th tuple $\langle l, r\rangle$ where l=1 (resp. l=0) means that the left philosopher may (resp. cannot) take this fork and l =? is the intermediate state where the philosopher is eating to later release the forks (thus abstracting the steps DP_1 and DP_2 in Example 4.3). The transition **Grab** (and **Grab-0** for $Phil_0$) abstracts the operational step from DP to DP_1 and **Grab-end** (and **Grab-0-end** for $Phil_0$) the transition from DP_1 to DP_3. Moreover, due to the definition of *Fork* in Example 4.2, from the state $\langle 0, 0\rangle$ the only possible transition for the fork is to reset its parameters leading to the state $\langle 1, 1\rangle$.

Next lemma proves that, for all philosopher P_i, *always is the case that* P_i *will eventually eat*. More precisely,

Lemma 4.1 (fairness) *Let* $n \geq 2$, $N \geq 1$ *and* $DP(n)$ *be the* n*-dining philosopher system formalized in Example 4.3. Then,* $\mathcal{O}(DP) \neq \emptyset$ *and, for all* $\sigma \in \mathcal{O}(DP)$*:*

(i) ***Deadlock-freeness****:* σ *is an infinite sequence ;*

(ii) ***Fairness****: for all* $i \in [0, n)$*, the label* ${}^{\tau}\backslash_{eat_i}$ *appears infinitely often in* σ*.*

Proof. Let us give a sketch of the proof (more details in the appendix and an automatic proof for a specific instance of n in the next section). For deadlock-freeness, consider the state $\mathcal{S}_n = \langle a_0, a_1\rangle\langle a_1, a_2\rangle\langle a_2, a_3\rangle \ldots \langle a_{n-1}, a_0\rangle$ where $a_i \in \{0, 1, ?\}$. If there is an a_i s.t. $a_i \in \{1, ?\}$, then, it is always possible to apply the rules **Grab** or **Grab-end** to make a transition. Otherwise (i.e., if $a_i = 0$ for all $i \in 0..n-1$) then, it is always possible to apply the **Reset** rule. The proof of fairness follows by contradiction. Let σ be a computation of $DP(n)$ such that $0 < m \leq n$ philosophers will never perform the eating action. This means that in σ, the state includes tuples of the form $\cdots\langle i, 1\rangle\langle 1, j\rangle\cdots$ where the 1's remain the same (i.e., these values never become 0 due to an application of **Grab**). We can show that σ must be finite and, indeed, there will be a maximum number of transitions before getting a deadlock (thus a contradiction since σ must be infinite). $\square$

A direct consequence of Lemma 4.1 is given by the following corollary.

Corollary 4.1 (Starvation freedom) *Let* $n \geq 2$, $N \geq 1$ *and* $DP(n)$ *be a* n*-dining philosopher system formalized as in Example 4.3. In every (infinite) computation, the* ***Grab-end*** *transition (the action after eating) occurs infinitely often for each adjacent tuple.*

Note that in Example 4.3 we can specify different constants N_i for the parameters of the process $Fork_i\langle l_i, r_i\rangle$, with the restriction that $r_i = l_{i^+_n}$, i.e. the number of times that two consecutive forks are used by their common adjacent philosopher must be the same. This is useful when we are modeling systems in which there are different priorities for the use of the resources. Also in situations where the network of agents is not completely balanced (since some of them may work faster than others).

In the next example, we show that values and constraints can be useful to specify, declaratively, the internal state of processes. For that, we consider the case where philosophers may decide to remain thinking for a while and then, they decide to eat. In this scenario, it is important for the system that philosophers moving to the thinking state do not block the activities of the others. We thus assume that philosophers take the decision (of thinking or eating again) and such decision must be communicated to the two adjacent forks. In turn, the forks may perform synchronizations with philosophers that have already consumed all their N usages whenever the adjacent philosopher is in the thinking state.

Example 4.4 (Constraints as states) *Consider the following specification for the philosophers:*

$$\begin{aligned}
\mathit{Phil}_i() &\stackrel{\text{def}}{=} {}^{idlL_i}\backslash_{idlR_{i^+_n}}.\mathit{Phil\text{-}Idle}_i\langle\rangle + {}^{upL_i}\backslash_{upR_{i^+_n}}.\mathit{Phil\text{-}Eat}_i\langle\rangle \\
\mathit{Phil\text{-}Eat}() &\stackrel{\text{def}}{=} {}^{\tau}\backslash_{eat_i}.{}^{dw_i}\backslash_{dw_{i^+_n}}.\mathit{Phil}_i\langle\rangle \\
\mathit{Phil\text{-}Idle}_i() &\stackrel{\text{def}}{=} {}^{tau}\backslash_{tk_i}.\mathit{Phil\text{-}Idle}_i\langle\rangle + {}^{wL_i}\backslash_{wR_i}.{}^{upL_i}\backslash_{upR_{i^+_n}}.\mathit{Phil\text{-}Eat}_i\langle\rangle
\end{aligned}$$

In Phil_i *we see two new sets of names and links:* $idlL_i$ *and* $idlR_i$ *(resp.* wL_i *and* wR_i*) are used to synchronize with both forks and communicate the fact that the philosopher goes to the thinking state (resp. starts eating again). The model for the forks is as follows:*

$$\begin{aligned}
\mathit{Fork}_i(l, r, idll, idlr) \stackrel{\text{def}}{=}\ & {}^{idlR_i}\backslash_{\tau}.\mathit{Fork}_i\langle l, r, idll, 1\rangle + {}^{\tau}\backslash_{idlL_i}.\mathit{Fork}_i\langle l, r, 1, idlr\rangle + \\
& {}^{wR_i}\backslash_{\tau}.\mathit{Fork}_i\langle l, r, idll, 0\rangle + {}^{\tau}\backslash_{wL_i}.\mathit{Fork}_i\langle l, r, 0, idlr\rangle + \\
& {}^{\tau}\backslash_{upL_i}(?l = 0 \wedge idlr = 1).{}^{\tau}\backslash_{dw_i}.\mathit{Fork}_i\langle l, r, idll, idlr\rangle + \\
& {}^{upR_i}\backslash_{\tau}(?r = 0 \wedge idll = 1).{}^{dw_i}\backslash_{\tau}.\mathit{Fork}_i\langle l, r, idll, idlr\rangle + \\
& \textit{the 3 choices in Example 4.2}
\end{aligned}$$

The process $DP(n)$ *is defined as usual, adding the new names in the set of restricted names.*

In Fork_i, besides l and r, we also have parameters to determine the current state of the left and right philosophers. The first line in the definition allows for communicating the decision of going to the thinking/idle state (for the right and left philosopher). Similarly, the second line is used to communicate the intention of start eating again. Most importantly, the third line allows for a synchronization with the left philosopher even if $l = 0$. In that case, the right philosopher must be in the idle state. Similarly for the forth line. In this system we cannot prove fairness as in Lemma 4.1 (since there are unfair always-*thinking* computations). We can assume (externally) a fairness condition ruling out such computations or specify a more controlled version of *Phil-Idle*$_i$ that, for instance, "counts" and controls the number of *thinking* actions.

Before closing this section, let us note that all the solutions presented here satisfy the usual requirements for this problem: *fully distribution* (there is no central agent coordinating the activities of the philosophers) and *symmetry* (all philosophers and forks are identical). The control of the agents defined here rely completely on their internal state and all of them are symmetrically defined as $Phil_i$ and $Fork_i$. If we dispense with symmetry, there is a simple solution for the problem in CCS where P_0 grabs first the fork on his left (F_0) and P_1 grabs first the fork on his right (again, F_0). Hence, there is no a deadlock in this asymmetric specification as the one described in the beginning of this section. As pointed out in [12], the solution for the problem in Multi-CCS (as well as ours in CCNA) is fully distributed in an abstract level: there is no a central shared memory. However, it is not possible to have a truly distributed deterministic implementation of this kind of multiparty synchronization mechanisms [15].

4.2 The network transportation system

The following example is a simplified version of the network transportation system presented in [7] where each transportation system has a specific cost. Passengers may specify a threshold for the value they are willing to pay for a trip starting and ending at the required stations. For simplicity, we are not considering the model of the stations. Here we model a complete trip of the passenger as a single transition that also records all the data concerning the trip (i.e. the sum of the costs of the used means of transport).

Example 4.5 (Network Transportation System) *Consider the following definitions:*

$$\begin{array}{lll} P = {}^{\tau}\backslash_{s_1} \mid {}^{s_3}\backslash_{\tau}(?\mathfrak{a} \le 5) & MoT(s1, s2, c) \triangleq {}^{s1}\backslash_{s2}\langle !c\rangle.MoT & \\ M1 = MoT(s1, s2, 3) & M2 = MoT(s2, s3, 2) & T1 = MoT(s1, s3, 7) \\ System = (\nu \boldsymbol{s})(P \mid M1 \mid M2 \mid T) & & \end{array}$$

Here, P is willing to go from s_1 to s_3. She offers its links for free but she constraints synchronizations to have cost at most 5. On the other side, $M1$ does not impose any constraint but it forces the final agreement to have cost at least 3. In this system, there is only one possible transition, namely,

$$System \xrightarrow{(\nu \boldsymbol{s})\, {}^{\tau}\backslash^{s1}_{s1}\backslash^{s2}_{s2}\backslash^{s3}_{s3}\backslash_{\tau}\langle !5\rangle(?5\le 5)} (\nu \boldsymbol{s})(M1 \mid M2 \mid T)$$

where P has to synchronize with both $M1$ and $M2$ (and pay 5). Note that P cannot take the train T since the chain ${}^{\tau}\backslash^{s1}_{s1}\backslash^{s3}_{s3}\backslash_{\tau}\langle !7\rangle(?7 \le 5)$ is not valid (Def. 3.10).

We can model the situation in which the passengers have two kinds of constraints: cost and time. In this case, values are tuples (Notation 3.9) of the form $\langle cost, time\rangle$. Each means of transport offers services at a given cost and speed and passengers may pose constraints on those values. Furthermore, adding a third element to the tuple, we can also restrict the number of connections a passenger is willing to accept (see Example 3.8).

4.3 Service Level Agreements (SLA)

We propose an extended model for the Service Level Agreements (SLA) protocol specified in [9]. In this kind of protocols, before the effective provisioning of a service, the involved parties should agree on a set of parameters, such as the cost the client should pay or the service quality the provider is willing to offer.

Example 4.6 (SLA Protocol) *Here we consider a client C asking a web hosting provider P the use of a service. P, in turn, can offer the service once it receives the availability of the bandwidth from a third party T. Hence we shall consider tuples of the form $\langle cost, bw\rangle$ (see Notation 3.9). The client is modeled as*

$$C \triangleq {}^{\tau}\backslash_{s}(?cost < Max_C \wedge bw > Min_B).C$$

where Max_C (maximal cost) and Min_B (minimal bandwidth) are constants for the model. We may have several T's offering different options for the provider, for instance:

$$T_1 \triangleq {}^{th}\backslash_{\tau}\langle !25, 100\rangle.T_1 \qquad\qquad T_2 \triangleq {}^{th}\backslash_{\tau}\langle !17, 70\rangle.T_2 + {}^{th}\backslash_{\tau}\langle !32, 130\rangle.T_2\,.$$

Here, T_1 has only one option of service (at cost 25) while T_2 offers two possibilities of bandwidth (70 *and* 130*) at different costs. The provider P charges an extra fee depending on the bandwidth availability that he has received from T:*

$$P \triangleq {}^{s}\backslash_{th}\langle !2,0\rangle(?cost < 60).P \qquad + {}^{s}\backslash_{th}\langle !3,0\rangle(?60 \le cost < 100).P \qquad + {}^{s}\backslash_{th}\langle !5,0\rangle(?cost \ge 100).P$$

The system is $SLA \triangleq (\nu\, s, th)(P \mid C \mid T_1 \mid T_2)$ and a possible transition is

$$SLA \xrightarrow{(\nu\, s,th)\,{}^{\tau}\backslash_{s}^{s}\backslash_{th}^{th}\backslash_{\tau}\langle !20\rangle(?c)} SLA$$

where $c = 20 < Max_C \wedge 70 > Min_B \wedge 60 \le 70 < 100$. *What we are observing is a synchronization between P, C and the first option of T_2 (and thus, the final cost is 20).* □

5 Concluding Remarks

On top of the tool described in [8], we have implemented a rewriting logic [16] specification of the semantics proposed here in the Maude System. The tool is available at https://gitlab.com/carlos_olarte/SiLVer. We built a suitable signature for the syntax of CCNA processes and specified the operational rules as rewriting rules. We profit from the symbolic techniques proposed in [8] to efficiently check when two or more links can be combined into a valid link chain. Using this tool, we can check for instance that for a particular instance of n, all the systems $DP(n)$ discussed in Section 4.1 are deadlock free. For that, it is just a matter to ask whether there is a reachable state without transition (i.e., a normal form, "$\Rightarrow !$"): `search` [1] `DP(2)` $\Rightarrow !$ `S:State` .. The answer is `No solution`. telling us that such state does not exist, thus proving deadlock freeness for $DP(2)$. More interestingly, we can verify the fairness condition in Lemma 4.1. For that, we use the model checker in Maude and ask if the property $\Box\Diamond\{{}^{\tau}\backslash_{eat_0}\}$ is valid. Here $\Box$ and $\Diamond$ are the linear time temporal logic (LTL) modalities "always" and "eventually". The answer is `true` for the system in Example 4.3 and `false` (with a suitable counterexample) for the other models. We can also verify *safety* for all the systems. For that, we can ask if there is a state reachable from $DP(2)$ where both ${}^{\tau}\backslash_{eat_0}$ and ${}^{\tau}\backslash_{eat_1}$ are enabled. The answer is `No solution`.. The tool also offers facilities to traverse the transition system, generate traces and produce a DOT file (graph description language) with the resulting transition system.

We are currently working on an extension of the Symbolic Link Modal Logic proposed in [8] to offer mechanisms to specify properties involving constraints. This should allow us to state properties such as "the process P cannot exhibit a n-party synchronization with $n > N$" or "the server will never admit more connections that its bandwidth-limit allows". Coupling this logic with the already implemented infrastructure for model checking in Maude will provide more (automatic) verification techniques to reason about CCNA specifications. It would be also interesting to explore a wider range of behavioral equivalences including weak-network-bisimulation and also stronger versions of network-bisimulations (where, e.g, $\bowtie$-related link chains are not identified). Efficient decision procedures for those equivalences have not been explored yet.

In the examples presented here, for the sake of uniformity, we have used only one CLM ($\mathcal{K}_{\mathbb{N}}$). There are many choices for it (see e.g., [11,3]). For instance, consider the structures $\mathcal{K}_P = \langle [0,1], \le, \times\rangle$ and $\mathcal{K}_F = \langle [0,1], \le, min\rangle$. In the first one, the subscript "P" is for *probability* and agents accumulate values in the real interval $[0,1]$ by multiplying them. Hence, the accumulated value gets closer to 0 when more agents are involved in an interaction. In the second structure the "F" stands for *fuzzy*, where values are accumulated by choosing the minimal value. In this case, agents can define a threshold for interaction based on their preferences expressed as values in the interval $[0,1]$. As pointed out in Lemma 3.3, it is possible to combine such structures to obtain richer ones. Some examples using those structures are in the tool's web page.

Multiparty calculi with different synchronization mechanisms have been proposed, e.g., in CSP [13] and full Lotos [22]. These calculi offer parallel operators that exhibit a set of action names (or channel names), and all the parallel processes offering that action (or an action along that channel) can synchronize by executing it. In [20], a binary form of input allows for a three-way communication. The reader may also refer to [14] where it is shown that CCS^n (or n-join CCS), an extension of CCS that allows prefixes to synchronize with at most n outputs, is strictly more expressive than CCS^{n-1}. The multiparty calculus most related to CNA is in [19], where links are named and are distinct from the usual input/output actions: there is one sender and one receiver (the output includes the final receiver name). Finally, we mention the cc-π calculus [9] that combines the name-passing discipline of the π-calculus with constraints in the style of concurrent constraint programming (see a survey in [21]). This calculus does not offer multiparty synchronization and its semantics is necessarily more involved due to the name-passing discipline of the π-calculus. As showed here, constraints in CCNA allows for a declarative control of processes in a very natural way.

Appendix

A Congruence (Theorem 3.16)

In this section we prove that $P \sim Q$ implies that, for any context $\mathcal{C}[\cdot]$, $\mathcal{C}[P] \sim \mathcal{C}[Q]$. Recall that the (co-inductive) technique for showing that two processes are (network) bisimilar consists in exhibiting a network bisimulation $\mathcal{R}$ containing the two processes (see, e.g., [23]). Since $P \sim Q$, we know that there exists a network bisimulation $\mathcal{R}$ containing the pair (P, Q). For each context $\mathcal{C}[\cdot]$, we show a suitable $\mathcal{R}'$ s.t. $(\mathcal{C}[P], \mathcal{C}[Q]) \in \mathcal{R}'$.

- Case $\mu.[\cdot]$. The needed relation is $\mathcal{R}' = \{(\mu.P, \mu.Q) \mid \mu \in VCC\} \cup \mathcal{R}$. Clearly, $\mu.P$ can only perform μ and proceed as P. Similarly for $\mu.Q$. Since $(P, Q) \in \mathcal{R}$, $\mathcal{R}'$ is indeed a network bisimulation.
- Case $[\cdot] + R$. Note that P and Q must be sequential processes. Let $\mathcal{R}' = \mathcal{R} \cup \{(P + R, Q + R) \mid R \in \mathcal{P}\} \cup \mathcal{I}$ where $\mathcal{I}$ is the identity relation on $\mathcal{P}$ ($\mathcal{P}$ is the set of CCNA processes). There are two possible transitions for $P + R$. (1) If $P + R \xrightarrow{\mu} P'$ then, it must be the case that $P \xrightarrow{\mu} P'$. Hence, there exists Q' and $\mu' \triangleright\!\triangleleft \mu$ s.t. $Q \xrightarrow{\mu'} Q'$. We conclude by noticing that $Q + R \xrightarrow{\mu'} Q'$ and $(P', Q') \in \mathcal{R}'$. (2) If R moves, we observe $P + R \xrightarrow{\mu} R'$. Then, $Q + R \xrightarrow{\mu} R'$ and clearly $(R', R') \in \mathcal{R}'$. The case $R + [\cdot]$ follows similarly.
- Case $[\cdot]|R$. The needed network bisimulation is $\mathcal{R}' = \{(P|R, Q|R) \mid (P, Q) \in \mathcal{R} \text{ and } R \in \mathcal{P}\}$. The process $P|R$ exhibits 3 kind of transitions. $P|R$ moves to $P'|R$ with label μ using the rule *Lpar*. Hence, $P \xrightarrow{\mu} P'$ and there exists Q' and $\mu' \triangleright\!\triangleleft \mu$ s.t. $Q \xrightarrow{\mu'} Q'$ and $(P', Q') \in \mathcal{R}$. By using *Lpar*, $Q|R \xrightarrow{\mu'} Q'|R$ and clearly $(P'|R, Q'|R) \in \mathcal{R}'$ as needed. The case when R moves (using *Rpar*) is trivial. If P and R synchronizes (using *Com*), it must be the case that $P \xrightarrow{\mu} P'$, $R \xrightarrow{\psi} R'$ and the label of the transition is $\mu \bullet \psi$. We also know that there exists Q' and $\xi \triangleright\!\triangleleft \mu$ s.t. $Q \xrightarrow{\xi} Q'$ and $(P', Q') \in \mathcal{R}$. We can suitable enlarge (via $\blacktriangleright\!\blacktriangleleft$ and $\triangleright\!\triangleleft$) the link chains in ψ and ξ to make $\psi \bullet \xi$ valid and $Q|R \xrightarrow{\xi \bullet \psi} Q'|R'$ as needed. The case $R|[\cdot]$ is similar.
- Case $(\nu a)[\cdot]$. The needed relation is $\mathcal{R}' = \{((\nu a)P, (\nu a)Q) \mid (P, Q) \in \mathcal{R} \text{ and } a \in VCC\}$. If $(\nu a)P \xrightarrow{(\nu a)\mu} (\nu a)P'$ it must be the case that $P \xrightarrow{\mu} P'$. Hence there exists Q' and $\mu' \triangleright\!\triangleleft \mu$ s.t. $Q \xrightarrow{\mu'} Q'$ and $(P', Q') \in \mathcal{R}$. We conclude by noticing that $(\nu a)Q \xrightarrow{(\nu a)\mu'} (\nu a)Q'$ (for that, we can easily show that if $(\nu a)\mu$ is valid, then $(\nu a)\mu'$ is also valid for $\mu' \triangleright\!\triangleleft \mu$.) and hence, $((\nu a)P', (\nu a)Q') \in \mathcal{R}'$ as needed. $\square$

B Proof of fairness (Lemma 4.1)

For the sake of readability, we shall change marginally the notation of the reachable states (processes) from DP(n). Note that the set of labels (ignoring the constraint $\mathtt{tt}$) of the transition system generate from DP is $L = \{{}^{\tau}\backslash_{eat_i}, {}^{\tau}\backslash_{tk_i} \mid i \in [0, n)\} \cup \{{}^{\tau}\backslash^{\tau}_{\tau}\backslash^{\tau}_{\tau}\backslash_{\tau}\} \cup \{{}^{\tau}\backslash_{\tau}\}$. The first component corresponds to the (visible) eating and thinking actions; the second component to the 3-party interaction for grabbing and releasing the forks (see transitions in Example 4.3); and ${}^{\tau}\backslash_{\tau}$ to the reset action (first line in the definition of Fork in Example 4.2). Let us use eat_i, tk_i, $grab_i$, $release_i$ and $reset_i$ to denote such actions. For conciseness, we fix $N = 1$.

The process $Fork_i(l, r)$ and its sucessor states will be represented as the tuple $\langle l, r\rangle$. For that, note that a fork can be in one of the following states (see Example 4.2):

- $Fork_i(1, r)$ (resp. $Fork_i(l, 1)$) where it can synchronize with the philosopher on the right (resp. on the left). We shall write these states, respectively, as $\langle 1, r\rangle$ and $\langle l, 1\rangle$.
- ${}^{\tau}\backslash_{dw_i}.Fork_i\langle 0, r\rangle$ and ${}^{dw_i}\backslash_{\tau}.Fork_i\langle l, 0\rangle$. The first (resp. second) is the result after a synchronization with the philosopher on the left (resp. right). Let us denote those states as the tuples $\langle ?, r\rangle$ and $\langle l, ?\rangle$.
- $Fork_i(0, 0)$ (notation $\langle 0, 0\rangle$) where the only possible transition is a reset action leading to $\langle 1, 1\rangle$.

We can also simplify the notation to represent the philosophers. For that, note that they can be in one of the following states (see processes and transitions in Example 4.3):

- $Phil_i$, where he can grab the forks or think. We shall use GT_i to denote that state.
- After thinking, the resulting process is $Phil'_i$ whose only possible action is to grab the forks. We shall denote that state as G_i.
- After grabbing the forks, the new state is ${}^{\tau}\backslash_{eat_i}.{}^{dw_i}\backslash_{dw_{i^+_n}}.Phil_i\langle\rangle$ where the only possible action is to eat. We shall use E_i to denote that state.
- After exhibiting the eat_i action, the new state is ${}^{dw_i}\backslash_{dw_{i^+_n}}.Phil$, from now on denoted as R_i.

- After releasing the forks, we are back in the state GT_i.

  Hence, any resulting process from $DP(n)$ can be succinctly represented as
  $P_0 \cdots P_i \cdots P_{n-1}\ \langle l_0, r_1\rangle\langle l_1, r_2\rangle\langle l_2, r_3\rangle \cdots\ \langle l_{n-1}, r_0\rangle$ where each P_i is either GT_i, G_i, E_i, or R_i.
  Some valid transitions of this system are:

(i) $P_0\ \cdots GT_i\ \cdots P_{n-1}\ \langle l_0, r_1\rangle \cdots \langle l_{i-1}, r_i\rangle\langle l_i, r_{i+1}\rangle \cdots \langle l_{n-1}, r_0\rangle \xrightarrow{tk_i} P_0\ \cdots G_i\ \cdots P_{n-1}\ \cdots$ (Phil i thinks)

(ii) $\cdots G_i \cdots\ \cdots \langle l_{i-1}, 1\rangle\langle 1, r_{i+1}\rangle \cdots \xrightarrow{grab_i} \cdots E_i\ \cdots\ \cdots \langle l_{i-1}, ?\rangle\langle ?, r_{i+1}\rangle \cdots$ (grabbing the forks)

(iii) $\cdots E_i\ \cdots\ \cdots \langle l_{i-1}, ?\rangle\langle ?, r_{i+1}\rangle \cdots \xrightarrow{eat_i} \cdots R_i\ \cdots\ \cdots \langle l_{i-1}, ?\rangle\langle ?, r_{i+1}\rangle \cdots$ (eating)

(iv) $\cdots R_i\ \cdots\cdots \langle l_{i-1}, ?\rangle\langle ?, r_{i+1}\rangle \cdots \xrightarrow{release_i} \cdots GT_i\ \cdots\ \cdots \langle l_{i-1}, 0\rangle\langle 0, r_{i+1}\rangle \cdots$ (release the forks).

(v) $\cdots\ \cdots \langle 0, 0\rangle \cdots \xrightarrow{reset_i} \cdots\ \cdots \langle 1, 1\rangle \cdots$ (reset the fork i)

After the transition (iv), the only available action for the i-th philosopher is to think (only once). In this state, he can only grab the forks again once his neighbors eat and the forks perform the reset action. More precisely, the configuration $\ldots\langle 1, 0\rangle\langle 0, 1\rangle\ldots$ means that the philosopher in the middle has eaten and, after thinking, he remains blocked until his neighbors eat and release the forks.

Proof. Lemma 4.1. We prove the two points (deadlock-freeness and fairness) separately.

(i) Consider the state $\mathcal{S}_n = P_0 \cdots P_{n-1}\ \langle a_0, a_1\rangle\langle a_1, a_2\rangle\langle a_2, a_3\rangle\ldots\langle a_{n\text{-}1}, a_0\rangle$ where $a_i \in \{0, 1, ?\}$. If there is an a_i s.t. $a_i \in \{1, ?\}$, then, it is always possible to exhibit a grab ($a_i = 1$), eat or release ($a_i =?$) transition. Otherwise (i.e., if $a_i = 0$ for all $i \in 0..n\text{-}1$) then, it is always possible to exhibit a reset transition.

(ii) The proof is by contradiction. Let $DP(n)$ be a n-dining philosopher system such that there exists $i \in [0, n)$ and $Phil_i$ that performs eat_i finitely often. Without loss of generality (due to the circularity of the configuration) assume that $i = 0$. Consider the suffix of the computation σ where $Phil_0$ has already performed all the eating actions, i.e., in the rest of the (infinite) computation, we do not observe eat_0. Hence, this philosopher is either in the state GT_0 and, after thinking, in state G_0. We shall show that this computation cannot be infinite (thus a contradiction).

Once the neighbor $Phil_1$ eats and releases the forks, we are in the following situation $DP' = G_0\ GT_1 \cdots \langle a_0, 0\rangle\langle 0, a_2\rangle\ldots$.

In this state, $Phil_1$ cannot eat again (he can only think). If $Phil_2$ eats, a_2 becomes 0 and a reset on $Fork_1$ is possible: $DP' \longrightarrow^* G_0\ GT_1 \cdots \langle a_0, 0\rangle\langle 0, 0\rangle\ldots \longrightarrow^* \xrightarrow{reset_1} G_0\ GT_1 \cdots \langle a_0, 0\rangle\langle 1, 1\rangle\ldots$

If $a_0 = 1$, $Fork_0$ cannot reset until $Phil_0$ eats (which is not possible by hypothesis). This reasoning goes on for all the $n - 1$ philosophers that are willing to eat. Once all of them have eaten, and all the forks that could have reset have already performed that action, the configuration is:

$$DP'' = G_0 \cdots G_{n-1}\langle 1, 0\rangle\langle 1, 0\rangle\ldots\langle 1, 0\rangle\langle 0, 1\rangle\langle 0, 1\rangle\ldots\langle 0, 1\rangle\langle 0, 1\rangle$$

Hence, if $Phil_0$ does not eat, at some point, all the philosophers will be blocked. In fact, counting only the eating and reset actions, the system can perform at most $\sum_{i=1}^{n\text{-}1} i = \frac{n\times(n-1)}{2}$ transitions:

$\langle 1, 1\rangle\langle 1, 1\rangle\langle 1, 1\rangle\ldots\langle 1, 1\rangle\langle 1, 1\rangle\langle 1, 1\rangle$	at most n-1 eat actions if $Phil_0$ does not eat
$\langle 1, 0\rangle\langle 0, 0\rangle\langle 0, 0\rangle\ldots\langle 0, 0\rangle\langle 0, 0\rangle\langle 0, 1\rangle$	at most n-2 reset actions
$\langle 1, 0\rangle\langle 1, 1\rangle\langle 1, 1\rangle\ldots\langle 1, 1\rangle\langle 1, 1\rangle\langle 0, 1\rangle$	at most n-3 eat actions
$\langle 1, 0\rangle\langle 1, 0\rangle\langle 0, 0\rangle\ldots\langle 0, 0\rangle\langle 0, 1\rangle\langle 0, 1\rangle$	at most n-4 reset actions
$\ldots$	
$\langle 1, 0\rangle\langle 1, 0\rangle\ldots\langle 1, 0\rangle\langle 0, 1\rangle\langle 0, 1\rangle\ldots\langle 0, 1\rangle\langle 0, 1\rangle$	no transition if $Phil_0$ does not eat

If $Phil_0$ does not eat, $Fork_0$ and $Fork_{n\text{-}1}$ cannot reset. Hence, from row 2 on, the state of these forks will be, respectively, $\langle 1, 0\rangle$ and $\langle 0, 1\rangle$. At this point, $Fork_1$ and $Fork_{n\text{-}2}$ are able to reset, and this justifies the configuration in row 3. At row 4, $Phil_1$ and $Phil_{n\text{-}2}$ cannot eat since the needed forks are not available and they cannot reset. Hence, assuming that $Phil_0$ does not eat breaks the possibility of restarting all the forks and a deadlock is reached.

□

References

[1] Abadi, M. and A. D. Gordon, *A calculus for cryptographic protocols: The spi calculus*, Inf. Comput. **148** (1999), pp. 1–70.
URL https://doi.org/10.1006/inco.1998.2740

[2] Bistarelli, S. and F. Gadducci, *Enhancing constraints manipulation in semiring-based formalisms*, in: G. Brewka, S. Coradeschi, A. Perini and P. Traverso, editors, *ECAI 2006*, Frontiers in Artificial Intelligence and Applications **141** (2006), pp. 63–67.
URL http://www.booksonline.iospress.nl/Content/View.aspx?piid=1647

[3] Bistarelli, S., U. Montanari and F. Rossi, *Semiring-based contstraint logic programming: syntax and semantics*, ACM Trans. Program. Lang. Syst. **23** (2001), pp. 1–29.
URL https://doi.org/10.1145/383721.383725

[4] Bodei, C., L. Brodo and R. Bruni, *Open multiparty interaction*, in: N. Martí-Oliet and M. Palomino, editors, *WADT 2012, Revised Selected Papers*, Lecture Notes in Computer Science **7841** (2012), pp. 1–23.
URL http://dx.doi.org/10.1007/978-3-642-37635-1_1

[5] Bodei, C., L. Brodo and R. Bruni, *A formal approach to open multiparty interactions*, Theor. Comput. Sci. **763** (2019), pp. 38–65.
URL https://doi.org/10.1016/j.tcs.2019.01.033

[6] Bodei, C., L. Brodo, R. Bruni and D. Chiarugi, *A flat process calculus for nested membrane interactions*, Sci. Ann. Comp. Sci. **24** (2014), pp. 91–136.
URL http://dx.doi.org/10.7561/SACS.2014.1.91

[7] Brodo, L. and C. Olarte, *Symbolic semantics for multiparty interactions in the link-calculus*, in: B. Steffen, C. Baier, M. van den Brand, J. Eder, M. Hinchey and T. Margaria, editors, *SOFSEM 2017*, LNCS **10139** (2017), pp. 62–75.
URL https://doi.org/10.1007/978-3-319-51963-0_6

[8] Brodo, L. and C. Olarte, *Verification techniques for a network algebra*, Fundam. Inform. **172** (2020), pp. 1–38.
URL https://doi.org/10.3233/FI-2020-1890

[9] Buscemi, M. G. and U. Montanari, *Cc-pi: A constraint-based language for specifying service level agreements*, in: R. De Nicola, editor, *ESOP 2007*, LNCS **4421** (2007), pp. 18–32.
URL https://doi.org/10.1007/978-3-540-71316-6_3

[10] Cardelli, L., *Brane calculi*, in: V. Danos and V. Schächter, editors, *CMSB 2004*, LNCS **3082** (2004), pp. 257–278.
URL https://doi.org/10.1007/978-3-540-25974-9_24

[11] Gadducci, F., F. Santini, L. F. Pino and F. D. Valencia, *Observational and behavioural equivalences for soft concurrent constraint programming*, J. Log. Algebr. Meth. Program. **92** (2017), pp. 45–63.
URL https://doi.org/10.1016/j.jlamp.2017.06.001

[12] Gorrieri, R. and C. Versari, "Introduction to Concurrency Theory - Transition Systems and CCS," Texts in Theoretical Computer Science. An EATCS Series, Springer, 2015.
URL https://doi.org/10.1007/978-3-319-21491-7

[13] Hoare, C. A. R., "Communications Sequential Processes," Prentice-Hall, Englewood Cliffs (NJ), USA, 1985.

[14] Laneve, C. and A. Vitale, *The expressive power of synchronizations*, in: *LICS 2010* (2010), pp. 382–391.
URL https://doi.org/10.1109/LICS.2010.15

[15] Lehmann, D. J. and M. O. Rabin, *On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem*, in: J. White, R. J. Lipton and P. C. Goldberg, editors, *POPL* (1981), pp. 133–138.
URL http://doi.acm.org/10.1145/567532.567547

[16] Meseguer, J., *Twenty years of rewriting logic*, J. Log. Algebr. Program. **81** (2012), pp. 721–781.
URL https://doi.org/10.1016/j.jlap.2012.06.003

[17] Milner, R., "Communication and Concurrency," International Series in Computer Science, Prentice Hall, 1989, sU Fisher Research 511/24.

[18] Milner, R., J. Parrow and D. Walker, *A calculus of mobile processes, I and II*, Inf. Comput. **100** (1992), pp. 1–40.
URL https://doi.org/10.1016/0890-5401(92)90008-4

[19] Montanari, U. and M. Sammartino, *Network conscious pi-calculus: a concurrent semantics*, in: *MFPS 2012*, Electronic Notes in Theoretical Computer Science 286 (2012), pp. 291–306.

[20] Nestmann, U., *On the expressive power of joint input*, Electronic Notes in Theoretical Computer Science **16(2)** (1998).

[21] Olarte, C., C. Rueda and F. D. Valencia, *Models and emerging trends of concurrent constraint programming*, Constraints An Int. J. **18** (2013), pp. 535–578.
URL https://doi.org/10.1007/s10601-013-9145-3

[22] Peter H. J. van Eijk, M. D., Chris A. Vissers, "The Formal Description Technique LOTOS," North-Holland, 1989.

[23] Sangiorgi, D., "Introduction to Bisimulation and Coinduction," Cambridge University Press, 2011.

[24] van Glabbeek, R. and P. Höfner, *Progress, justness, and fairness*, ACM Comput. Surv. **52** (2019), pp. 69:1–69:38.
URL https://doi.org/10.1145/3329125

An Investigation of Linear Substitution λ-Calculus as Session-Typed Processes

S. Alves
DCC-FCUP & CRACS
University of Porto
Porto, Portugal

D. Nantes-Sobrinho
Departamento de Matemática
Universidade de Brasília
Brasília, Brazil

J.A. Pérez
Bernoulli Institute
University of Groningen
The Netherlands

D. Ventura
Instituto de Informática
Universidade Federal de Goiás
Goiânia, Brazil

Abstract

This work in progress investigates a type directed encoding of the *simply-typed* linear substitution λ-calculus to session-typed π-processes, taking into account an operational semantics defined by a linear weak head reduction strategy. The encoding adapts the one by Toninho et. al., which encodes the simply-typed λ-calculus into a session-typed π-calculus through linear logic, while extending it to explicit substitutions, considering a generalisation of the π-calculus operational semantics with reduction rules at *a distance*, as proposed by Accattoli. These changes, while weakening the simulation of λ-calculus in the π-calculus, when compared with the 1-to-1 simulation obtained by Accattoli, allow for a tight encoding of well-typed functions as session-typed processes.

Keywords: Lambda Calculus, Pi-Calculus, Encoding, Session Types, Simulation

1 Introduction

This paper proposes an in-depth investigation on encodings from *functions to processes*, focusing on works proposed by Accattoli [1] and by Toninho et. al. [13].

Ever since Milner's first encoding of the λ-calculus [9], motivated by the desire to establish a canonical model for the concurrent paradigm in the same way that the λ-calculus provides a canonical model for the functional paradigm, there have been a variety of subsequent encodings for the various extensions of the π-calculus, like the ones in [10,11,12,5,4], just to mention a few.

Inspired by Milner's encodings of the λ-calculus, Accattoli [1] proposed a tight correspondence between a λ-calculus with linear substitutions and the π-calculus, giving a strong simulation with respect to reduction. In order to obtain such an encoding, explicit substitutions *at a distance* were used, however, only the untyped version of λ_{lsub} was considered. In the realm of simply typed terms, Toninho et. al [13], explored the relation between simply typed terms and session typed processes, through yet another encoding on functions as processes based on a linear encoding of terms in linear logic.

In this paper, we combine both approaches to obtain a type-directed encoding of a linear substitution λ-calculus, named λ_{lsub}-calculus, into session typed π-processes, in which the reduction strategy is the *linear weak head reduction* that corresponds closely to evaluation in the π-calculus. The idea is to explore the qualities of both works, to have an encoding in which the simulation of λ-calculus in the π-calculus is as tight as it can be, as in [1]; on the other hand, we also wish to maintain the correspondence between well-typed functions as session-typed processes, as in [13]. There is however a mismatch between their encodings: while the former considers the linear substitution λ_{lsub} calculus with explicit substitutions at a distance and an asynchronous fragment of the π-calculus with unary and binary input and outputs, the latter works with simply typed linear λ-terms, and an synchronous fragment of the π-calculus, which contains only unary input/output in addition to the *forward construct* $[x \leftrightarrow y]$ that equates channel names x and y.

Two approaches were considered: to use Acattoli's encoding and operational semantics, and develop a new strategy to connect the simply typed λ_{lsub} to session typed processes; an alternative is to extend the encoding by Toninho et. al., which already has the connection with session types, by adding explicit substitutions at a distance and investigate what is obtained from the weak head linear strategy of reduction. In this work we have opted for the latter approach. Although we could inherit a lot from both works, the small changes made on the encoding and on the operational semantics caused interesting modifications in the final result. To begin, we consider a simple types discipline for λ_{lsub}. To follow the weak head linear reduction strategy we extended the notions of evaluation/substitution contexts to linear λ-calculus. A translation $[\![\cdot]\!]_a$ from typed λ_{lsub}-calculus to the typed linear λ-calculus is extended with contexts and explicit substitutions. Technical results regarding operational correspondence and dynamic soundness are presented, it is worthy mentioning that their proofs follow the techniques in [1], due to the semantics involved. The encoding proposed in [13] now extends to explicit substitutions; therefore, all the intermediate translations that the authors used had to be extended to deal with explicit substitutions. The simulation results are not as tight as initially expected (Theorems 4.4 and 4.7). This is because of the forward construct, which induces some extra computation steps, resulting in a scenario where one λ-reduction step corresponds to three π-reduction steps.

Organisation.
Section 2 introduces the syntax and semantics of linear substitution calculus λ_{lsub}. Some properties of the calculus are given. Section 3 gives the fragment of the π-calculus that we will work with, operational semantics and some syntactic properties. In Section 4 we give the encoding from λ_{lsub} to π-calculus and then some simulation results. Finally, Section 5 concludes the paper.

2 Preliminaries

In this section we present a simply typed version of the *linear substitution calculus* λ_{lsub}. Following the steps of [13], a *linear version* was used as an intermediate language in obtaining the encoding to session typed processes. For the sake of clarity, the corresponding $\hat{\lambda}_{lsub}$-calculus is included in the full report [2]. Basic notions of the λ-calculus are assumed, for more details we refer the reader to [3].

2.1 *Simply typed linear substitution calculus λ_{lsub}*

λ_{lsub} is given by the following grammar of terms M, N and types τ, σ, ρ, where α denotes *base types*. Typing judgments have the form $\Gamma \vdash M : \tau$, where the typing context Γ consists of a finite set of (term) variable type assignments of the form $x : \sigma$, where any variable occurs at most once as subject. *Typing derivation rules* are given in Figure 1.

$$\begin{array}{ll}\text{(Terms)} & M, N ::= \; x \mid \lambda x.M \mid MN \mid M[N/x] \\ \text{(Types)} & \tau, \sigma, \rho ::= \alpha \mid \sigma \to \tau\end{array}$$

The constructor $M[N/x]$ is called *explicit substitution* (of N for x in M), the implicit substitution is denoted by $M\{N/x\}$. The set $\mathtt{fv}(M)$ of free variables of M is defined as usual where, in particular, $\mathtt{fv}(M[N/x]) = (\mathtt{fv}(M) \setminus \{x\}) \cup \mathtt{fv}(N)$.

$$\frac{}{\Gamma, x{:}\tau \vdash x : \tau}\,(\mathtt{ax}) \qquad \frac{\Gamma \vdash N : \sigma \qquad \Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash M[N/x] : \tau}\,(\mathtt{cut})$$

$$\frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}\,(\to \mathtt{i}) \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\,(\to \mathtt{e})$$

Fig. 1. Typing Rules for λ_{lsub}

Contexts and Operational Semantics for λ_{lsub}.
An *evaluation context*, denoted by $\mathbb{E}_S$, is defined according to the following grammar:

$$\mathbb{E}_\emptyset := (\![\,\cdot\,]\!) \mid \mathbb{E}_\emptyset M \qquad \mathbb{E}_{S \uplus \{x\}} := \mathbb{E}_S[M/x] \mid \mathbb{E}_{S \uplus \{x\}} M$$

A *substitution context*, denoted by $\mathbb{L}_S$, is given by:

$$\mathbb{L}_\emptyset := (\![\,\cdot\,]\!) \qquad \mathbb{L}_{S \uplus \{x\}} := \mathbb{L}_S[M/x]$$

In both cases, S denotes the set of variables captured by the context. Evaluation contexts will range over $\mathbb{E}, \mathbb{E}', \mathbb{E}'', \ldots$ whereas substitution contexts will range over $\mathbb{L}, \mathbb{L}', \mathbb{L}'', \ldots$. For instance, the list context $(\![\,\cdot\,]\!)[M_1/x][M_2/y]$ can be denoted by $\mathbb{L}_{\{x,y\}}(\![\,\cdot\,]\!)$, where $\mathbb{L}_{\{x,y\}}(\![N]\!)$ denotes $N[M_1/x][M_2/y]$.

Operational semantics is defined by the *linear weak head reduction* $\multimap$ given as the union of closure by evaluation contexts of the rules $\multimap_{dB}$ and $\multimap_{ls}$:

$$\mathbb{L}_S(\!\![\lambda x.M]\!\!)N \multimap_{dB} \mathbb{L}_S(\!\![M[N/x]]\!\!) \qquad \mathbb{E}_S(\!\![x]\!\!)[N/x] \multimap_{ls} \mathbb{E}_S(\!\![N]\!\!)[N/x],\ \text{if } x \notin S$$

Both rules assume that $\mathtt{fv}(N) \cap S = \emptyset$.

In order to establish soundness of type assignments, proving the subject reduction property, a linear) substitution lemma is proved beforehand as usual, being a particular case of partial substitutions[8].

Lemma 2.1 (Subject Reduction) *If $\Gamma \vdash M : \tau$ and $M \multimap M'$ then $\Gamma \vdash M' : \tau$.*

Proof. By induction on $\multimap$, with a Linear Substitution Lemma. □

3 The π-calculus

We consider a fragment of synchronous π-calculus without sums, with the forwarding process:

$$P, Q \quad ::= x\langle y\rangle.P \mid (\nu x)P \mid x(y).P \mid !x(y).P \mid (P \mid Q) \mid [x \leftrightarrow y]$$

The set $\mathtt{fn}(P)$ of free name of a process P is defined as expected, considering (νx) and $x(y)$ as binders. In other words, $\mathtt{fn}((\nu x)P) = \mathtt{fn}(P) \setminus \{x\}$ and $\mathtt{fn}(x(y).P) = (\mathtt{fn}(P) \setminus \{y\}) \cup \{x\}$.

A *non-blocking context*, range over $\mathbb{N}, \mathbb{N}', \mathbb{N}'', \ldots$, is also parameterized by a set S of variables, and defined by:

$$\mathbb{N}_\emptyset := (\!\![\cdot]\!\!) \mid (\mathbb{N}_\emptyset \mid Q) \mid (P \mid \mathbb{N}_\emptyset) \qquad \mathbb{N}_{S \uplus \{x\}} := (\nu x)\mathbb{N}_S \mid \mathbb{N}_\emptyset(\!\![\mathbb{N}_{S\uplus\{x\}}]\!\!)$$

Structural congruence $\equiv$ is the least congruence generated by the following rules and closed by non-blocking contexts.

$$\begin{array}{rclcrclcrcl} P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & & (\nu x)(\nu y)P & \equiv & (\nu y)(\nu x)P & & P \mid Q & \equiv & Q \mid P \\ x \notin \mathtt{fn}(P) \Rightarrow P \mid (\nu x)Q & \equiv & (\nu x)(P \mid Q) & & P \equiv_\alpha Q & \Rightarrow & P \equiv Q & & [x \leftrightarrow y] & \equiv & [y \leftrightarrow x] \end{array}$$

Operational Semantics

The *reduction relation* $\rightarrow_\pi$ is defined by the rewriting rules, closed by non-blocking contexts modulo $\equiv$:

$$\begin{array}{rcl} x\langle y\rangle.Q \mid x(z).P & \rightarrow_\tau & Q \mid P\{y/z\} \\ x\langle y\rangle.Q \mid !x(z).P & \rightarrow_{!\tau} & Q \mid P\{y/z\} \mid !x(z).P \\ (\nu x)(P \mid [x \leftrightarrow y]) & \rightarrow_r & P\{y/x\} \quad (\text{provided } x \neq y) \end{array}$$

Pi-calculus at a distance.

Consider the reduction rule $\Rightarrow_\pi$ given by the closure by non-blocking contexts of the relation: if $x \notin S \cup V \cup W$, then

$$\begin{array}{rcl} \mathbb{N}_S(\!\![x\langle y\rangle.Q]\!\!) \mid \mathbb{N}'_V(\!\![x(z).P]\!\!) & \Rightarrow_\tau & \mathbb{N}'_V(\!\![\mathbb{N}_S(\!\![Q \mid P\{y/z\}]\!\!)]\!\!) \\ \mathbb{N}_S(\!\![x\langle y\rangle.Q]\!\!) \mid \mathbb{N}'_V(\!\![!x(z).P]\!\!) & \Rightarrow_{!\tau} & \mathbb{N}'_V(\!\![\mathbb{N}_S(\!\![Q \mid P\{y/z\} \mid !x(z).P]\!\!)]\!\!) \\ \mathbb{N}_{S\uplus\{x\}}(\!\![\mathbb{N}'_V(\!\![P]\!\!) \mid \mathbb{N}''_W(\!\![[x \leftrightarrow y]]\!\!)]\!\!) & \Rightarrow_r & \mathbb{N}_S(\!\![\mathbb{N}'_V(\!\![\mathbb{N}''_W(\!\![P]\!\!)]\!\!)\{y/x\}]\!\!) \end{array}$$

The reduction rules in $\Rightarrow_\pi$ take in account the forward construct. In addition, there are conditions on the variables which are implicit from the Barendregt's Convention on bound variables: $S \cap V = \emptyset$, $S \cap \mathtt{fn}(P) = \emptyset$ and $\mathtt{fn}(\mathbb{N}_S) \cap V = \emptyset$.

Lemma 3.1 *Let S be a set of variables, $\mathbb{N}_S$ a non-blocking context, P a process such that $\mathtt{fn}(P) \cap S = \emptyset$, and $x, y \notin S$. Then*

(i) $\mathbb{N}_S(\!\![Q]\!\!) \mid P \equiv \mathbb{N}_S(\!\![Q \mid P]\!\!)$.

(ii) *If $x \notin \mathtt{fn}(\mathbb{N}_S)$ then $(\nu x)\mathbb{N}_S(\!\![P]\!\!) \equiv \mathbb{N}_S(\!\![(\nu x)P]\!\!)$.*

Lemma 3.2 (Harmony Lemma) *Let P and Q be processes:*

$$\frac{}{\Gamma; x{:}\tau \Rightarrow [x \leftrightarrow a] :: a : \tau}\ (\texttt{id}) \qquad \frac{\Gamma; \Delta, x{:}\sigma \Rightarrow P :: a : \tau}{\Gamma; \Delta \Rightarrow a(x).P :: a : \sigma \multimap \tau}\ (\multimap \texttt{R})$$

$$\frac{\Gamma; \emptyset \Rightarrow P :: b : \tau}{\Gamma; \emptyset \Rightarrow !a(b).P :: a : !\tau}\ (\texttt{!R}) \qquad \frac{\Gamma, u : \sigma; \Delta \Rightarrow P :: a : \rho}{\Gamma; \Delta, b : !\sigma \Rightarrow P\{b/u\} :: a : \rho}\ (\texttt{!L})$$

$$\frac{\Gamma; \Delta \Rightarrow P :: b : \sigma \qquad \Gamma; \Delta', x : \tau \Rightarrow Q :: a : \rho}{\Gamma; \Delta, \Delta', x : \sigma \multimap \tau \Rightarrow (\nu b)x\langle b\rangle.(P \mid Q) :: a : \rho}\ (\multimap \texttt{L}) \qquad \frac{\Gamma, u : \sigma; \Delta, x : \sigma \Rightarrow P :: a : \rho}{\Gamma, u : \sigma; \Delta \Rightarrow (\nu x)u\langle x\rangle.P :: a : \rho}\ (\texttt{copy})$$

$$\frac{\Gamma; \Delta \Rightarrow P :: b : \sigma \qquad \Gamma; \Delta', x{:}\sigma \Rightarrow Q :: a : \tau}{\Gamma; \Delta, \Delta' \Rightarrow (\nu b)(P \mid Q) :: a : \tau}\ (\texttt{cut}) \qquad \frac{\Gamma; \emptyset \Rightarrow P :: b : \sigma \qquad \Gamma, u : \sigma; \Delta \Rightarrow Q :: a : \tau}{\Gamma; \Delta \Rightarrow (\nu u)(!u\langle b\rangle.P \mid Q) :: a : \tau}\ (\texttt{cut}^!)$$

Fig. 2. Session Typing Rules

(i) *There exists P' such that $P \equiv P' \Rightarrow_\pi Q$ iff there exists Q' such that $P \Rightarrow_\pi Q' \equiv Q$.*

(ii) *$P \rightarrow_\pi Q$ iff there exists Q' such that $P \Rightarrow_\pi Q' \equiv Q$.*

4 Functions as Session Typed Processes

In the present section we relate simply typed terms in the λ_{lsub}-calculus to its corresponding encoding as a session typed process. The encoding is derived following [13], in which intermediate steps are omitted. It is worth noticing that, in an encoding of λ_{lsub} in a linear calculus, named $\hat{\lambda}_{lsub}$, explicit substitution translations were understood through terms already present in the linear calculus of [13] and evaluation/list contexts encodings were straightforward. Having said that, the translation from $\hat{\lambda}_{lsub}$ to session typed processes has no essential difference from [13] (see [2] for further details).

Remark 1 *As for the original encoding of the λ-calculus into the π-calculus given by Milner, the encoding of λ_{lsub} into π-calculus is parameterized by a channel name a as in [1] we will assume that these special names are taken from a set $\mathcal{A}$ which is disjoint from the set of variable names V, and whose elements are denoted by $a, b, c, \ldots$.*

A typing judgement for processes is of the form $\Gamma; \Delta \Rightarrow P :: a : A$, where P is a process that implements a session of type A on channel a, based on assignments of types to channel names given by the contexts Γ and Δ. Channel names in Γ, Δ and a must be mutually distinct, renaming of bound names are assumed when necessary. The standard session typing rules are given in Figure 2 are taken from [6].

4.1 Encoding of λ_{lsub} into π-calculus

The encoding $[\![\cdot]\!]_a$ from λ_{lsub} to π-calculus below, is basically the same encoding $[\![\cdot]\!]_z$ given in [13], extended with explicit substitutions [1] .

$$\begin{aligned}
[\![x]\!]_a &= (\nu y)u_x\langle y\rangle.[y \leftrightarrow a] \\
[\![\lambda x.M]\!]_a &= a(x).(\nu y)([x \leftrightarrow y] \mid [\![M]\!]_a\{y/u_x\}) \\
[\![MN]\!]_a &= (\nu b)([\![M]\!]_b \mid (\nu c)b\langle c\rangle.((!c(y).[\![N]\!]_y) \mid [b \leftrightarrow a])) \\
[\![M[N/x]]\!]_a &= (\nu b)([\![M]\!]_a\{b/u_x\} \mid !b(c).[\![N]\!]_c)
\end{aligned}$$

The example below illustrates the encoding of the λ-term $(\lambda x.M)N$ as well as some reduction steps:

[1] variables named u_x correspond to unrestricted variables in the $\hat{\lambda}_{lsub}$ linear calculus, encoding a term variable x from λ_{lsub}

Example 4.1 For reduction:

$$\begin{array}{rcll}
[\![(\lambda x.M)N]\!]_a & = & (\nu b)([\![\lambda x.M]\!]_b \mid (\nu y)b\langle y\rangle.((!y(c).[\![N]\!]_c) \mid [b \leftrightarrow a])) & (1)\\
& = & (\nu b)(b(x).(\nu y')([x \leftrightarrow y'] \mid [\![M]\!]_b\{y'/u_x\}) \mid (\nu y)b\langle y\rangle.((!y(c).[\![N]\!]_c) \mid [b \leftrightarrow a])) & (2)\\
& \equiv & (\nu b)(\nu y)(b(x).(\nu y')([x \leftrightarrow y'] \mid [\![M]\!]_b\{y'/u_x\}) \mid b\langle y\rangle.((!y(c).[\![N]\!]_c) \mid [b \leftrightarrow a])) & (3)\\
& \equiv & (\nu b)(\nu y)(b\langle y\rangle.((!y(c).[\![N]\!]_c) \mid [b \leftrightarrow a]) \mid b(x).(\nu y')([x \leftrightarrow y'] \mid [\![M]\!]_b\{y'/u_x\})) & (4)\\
& \Rightarrow_\tau & (\nu b)(\nu y)((!y(c).[\![N]\!]_c) \mid [b \leftrightarrow a] \mid (\nu y')([y \leftrightarrow y'] \mid [\![M]\!]_b\{y'/u_x\})) & (5)\\
& \equiv & (\nu y)((!y(c).[\![N]\!]_c) \mid (\nu b)((\nu y')([y \leftrightarrow y'] \mid [\![M]\!]_b\{y'/u_x\}) \mid [b \leftrightarrow a])) & (6)\\
& \equiv & (\nu y)(\nu b)(\nu y')((!y(c).[\![N]\!]_c) \mid [y \leftrightarrow y'] \mid [\![M]\!]_b\{y'/u_x\} \mid [b \leftrightarrow a]) & (7)\\
& \equiv & (\nu y')((\nu y)((!y(c).[\![N]\!]_c) \mid [y \leftrightarrow y']) \mid (\nu b)([\![M]\!]_b\{y'/u_x\} \mid [b \leftrightarrow a])) & (8)\\
& \Rightarrow_r & (\nu y')((!y'(c).[\![N]\!]_c) \mid (\nu b)([\![M]\!]_b\{y'/u_x\} \mid [b \leftrightarrow a])) & (9)\\
& \Rightarrow_r & (\nu y')((!y'(c).[\![N]\!]_c) \mid [\![M]\!]_a\{y'/u_x\}) & (10)
\end{array}$$

As expected, the encoding preserves typability.

Theorem 4.2 *If* $\Gamma \vdash M : \tau$ *then* $\langle\!\langle\Gamma\rangle\!\rangle; \emptyset \Rightarrow [\![M]\!]_a :: a : \langle\!\langle\tau\rangle\!\rangle$

Proof. Induction on $\Gamma \vdash M : \tau$. We present the case for $(\texttt{cut})$ rule: $\dfrac{\Gamma \vdash N : \sigma \qquad x{:}\sigma, \Gamma \vdash M : \tau}{\Gamma \vdash M[N/x] : \tau}$

By *i.h.* we have both $\langle\!\langle\Gamma\rangle\!\rangle; \emptyset \Rightarrow [\![N]\!]_b :: b : \langle\!\langle\sigma\rangle\!\rangle$ and $u_x{:}\langle\!\langle\sigma\rangle\!\rangle, \langle\!\langle\Gamma\rangle\!\rangle; \emptyset \Rightarrow [\![M]\!]_a :: a : \langle\!\langle\tau\rangle\!\rangle$. Therefore,

$$\dfrac{\dfrac{\langle\!\langle\Gamma\rangle\!\rangle; \emptyset \Rightarrow [\![N]\!]_b :: b : \langle\!\langle\sigma\rangle\!\rangle}{\langle\!\langle\Gamma\rangle\!\rangle; \emptyset \Rightarrow !y(b).[\![N]\!]_b :: b : !\langle\!\langle\sigma\rangle\!\rangle} \qquad \dfrac{u_x{:}\langle\!\langle\sigma\rangle\!\rangle, \langle\!\langle\Gamma\rangle\!\rangle; \emptyset \Rightarrow [\![M]\!]_a :: a : \langle\!\langle\tau\rangle\!\rangle}{\langle\!\langle\Gamma\rangle\!\rangle; y : !\langle\!\langle\sigma\rangle\!\rangle \Rightarrow [\![M]\!]_a\{y/u_x\} :: a : \langle\!\langle\tau\rangle\!\rangle}}{\langle\!\langle\Gamma\rangle\!\rangle; \emptyset \Rightarrow (\nu y)(!y(b).[\![N]\!]_b \mid [\![M]\!]_a\{y/u_x\}) :: a : \langle\!\langle\tau\rangle\!\rangle}$$

□

Regarding subject reduction, not all terms in structural-equivalent classes of terms are typable. We illustrate this issue in the example below.

Example 4.3 [Cont. Example 4.1]

Let $\Gamma \vdash (\lambda x.M)N : \tau$. Then $\langle\!\langle\Gamma\rangle\!\rangle; \emptyset \Rightarrow (\nu b)([\![\lambda x.M]\!]_b \mid (\nu y)b\langle y\rangle.((!y(c).[\![N]\!]_c) \mid [b \leftrightarrow a])) :: a : \langle\!\langle\tau\rangle\!\rangle$, from Thm. 4.2, where $[\![(\lambda x.M)N]\!]_a = (\nu b)([\![\lambda x.M]\!]_b \mid (\nu y)b\langle y\rangle.((!y(c).[\![N]\!]_c) \mid [b \leftrightarrow a]))$ is the term in (1)/(2).

When reducing from term in (4) to term in (5), the latter is not typable in the session system. On the other hand, term in (6) is typable in the system:

$$\langle\!\langle\Gamma\rangle\!\rangle; \emptyset \Rightarrow (\nu y)((!y(c).[\![N]\!]_c) \mid (\nu b)((\nu y')([y \leftrightarrow y'] \mid [\![M]\!]_b\{y'/u_x\}) \mid [b \leftrightarrow a])) :: a : \langle\!\langle\tau\rangle\!\rangle$$

.

4.2 Simulation Results

In this subsection we show that simulation of one step in the λ_{lsub}-calculus demands exactly 3 steps of the π-calculus in the corresponding encoding (Theorems 4.4 and 4.7), while completeness is obtained directly instead of relying in some sort of *substitution lifting* as introduced in [13] [2]. In order to obtain such results, technicalities regarding the interaction between contexts and reduction had to be proven. Complete proofs are available in [2].

Theorem 4.4 ($\rightarrow_\pi$ 3-simulates $\multimap$ via $[\![]\!]_a$)

(i) $M' \multimap_{\texttt{dB}} N$ *implies* $[\![M']\!]_a \Rightarrow_\tau \overset{2}{\Rightarrow}_r \equiv [\![N]\!]_a$.

(ii) $M' \multimap_{\texttt{ls}} N$ *implies* $[\![M']\!]_a \Rightarrow_{!\tau} \Rightarrow_r \equiv [\![N]\!]_a$.

Proof.

[2] Definition 3.2 and Theorem 3.3 of [13]

(i) Then M' is of the form $M' = \mathbb{L}_S(\!|\lambda x.M|\!)N$ and the proof is divided in two cases, depending on the substitution context $\mathbb{L}_S$. For instance, when $\mathbb{L}_S = (\!|\cdot|\!)$, then $M' = (\lambda x.M)N$ and the reduction occurs in the head, that is, $M' = (\lambda x.M)N \multimap_{\mathtt{dB}} M[N/x]$. Notice that

$$\begin{aligned}
[\![(\lambda x.M)N]\!]_a &= (\nu b)([\![\lambda x.M]\!]_b \mid (\nu y_0)b\langle y_0\rangle.((!y_0(x).[\![N]\!]_x) \mid [b \leftrightarrow a])) \\
&= (\nu b)(b(x).(\nu y_1)([x \leftrightarrow y_1] \mid [\![M]\!]_b\{y_1/u_x\}) \mid (\nu y_0)b\langle y_0\rangle.((!y_0(x).[\![N]\!]_x) \mid [b \leftrightarrow a])) \\
&\equiv (\nu b y_0)(b(x).(\nu y_1)([x \leftrightarrow y_1] \mid [\![M]\!]_b\{y_1/u_x\}) \mid b\langle y_0\rangle.((!y_0(x).[\![N]\!]_x) \mid [b \leftrightarrow a])) \\
&\Rightarrow_\tau (\nu b y_0)((\nu y_1)([y_0 \leftrightarrow y_1] \mid [\![M]\!]_b\{y_1/u_x\} \mid ((!y_0(x).[\![N]\!]_x) \mid [b \leftrightarrow a])) \\
&\Rightarrow_r^2 (\nu y_0)([\![M]\!]_a\{y_0/u_x\} \; !y_0(x).[\![N]\!]_x) = [\![M[N/x]]\!]_a
\end{aligned}$$

The other cases can be found in the longer version of this paper [2].

□

Proposition 4.5 *There is no M such that $[\![M]\!]_a \Rightarrow_r Q$, for some Q.*

Proof. The proof follows by a direct inspection on the rule $\Rightarrow_r$. □

The next result adapts Lemma 7 in [1] to the proposed encoding and gives a characterisation of λ_{lsub}-terms whose encodings have some specific fragments, this will allow to work on simulation steps.

Lemma 4.6 *Let V and S a set of variable names and a set of special names, respectively.*

(i) *If $[\![M]\!]_a = \mathbb{N}_{V\uplus S}(\!|a(x).P|\!)$ with $a \notin S$ then $S = \emptyset$, $P = (\nu y)([x \leftrightarrow y]) \mid Q\{y/u_x\})$ and there exist N and $\mathbb{L}_V$ such that $Q = [\![N]\!]_a$ and $M = \mathbb{L}_V(\!|\lambda x.N|\!)$.*

(ii) *If $[\![M]\!]_a = \mathbb{N}_{V\uplus S}(\!|u_x\langle y\rangle.P|\!)$ with $y \in V$, and $u_x \notin V$ an unrestricted variable, then $P = [y \leftrightarrow a]$ and there exist $\Sigma \subseteq V$ and $\mathbb{E}_\Sigma$ such that $M = \mathbb{E}_\Sigma(\!|x|\!)$ (and $x \notin \Sigma$). Moreover, $\mathbb{N}_{V\uplus S} = (\nu y)(\mathbb{N}_{V'\uplus S})$ where $V = V' \uplus \{y\}$.*

Proof. In both cases the proof follows by induction on M. The full proof can be found in [2].

□

The theorem below shows that $\multimap$ simulates π but not so strongly as in [1], here we have a 1 for 3 relation, that is, one step of $\multimap$ corresponds to one step of $\Rightarrow_\tau$ followed by two renaming steps $\Rightarrow_r$.

Theorem 4.7 ($\multimap$ 3-simulates $\Rightarrow_\pi$ via $[\![]\!]_a$)

a) If $[\![M]\!]_a \Rightarrow_\tau Q$ then there exists N such that $M \multimap_{dB} N$ and $Q \Rightarrow_r^2\equiv [\![N]\!]_a$.

b) If $[\![M]\!]_a \Rightarrow_{!\tau} Q$ then there exists N such that $M \multimap_{ls} N$ and $Q \Rightarrow_r\equiv [\![N]\!]_a$.

Proof. In both cases, the proof is by induction on the structure of M. The interesting case happens when $M = N[L/z]$ and the reduction occurs at the root. Then $[\![M]\!]_a = (\nu b)([\![N]\!]_a\{b/u_z\} \; !b(c).[\![L]\!]_c)$ and we have to analyse the following cases:

a) The reduction $\Rightarrow_\tau$ cannot happen at the root.

b) In order to obtain such a reduction, $[\![N]\!]_a = \mathbb{N}_{V\uplus S}(\!|u_z\langle y\rangle.P|\!)$. By Lemma 4.6, $u_z \notin V$, $P = [y \leftrightarrow a]$ and $N = \mathbb{E}_\Sigma(\!|z|\!)$ for some context $\mathbb{E}_\Sigma$ and $\Sigma \subset V$. Besides, $\mathbb{N}_{V\uplus S} = (\nu y)(\mathbb{N}_{V'\uplus S})$, therefore,

$$\begin{aligned}
[\![M]\!]_a &= (\nu b)(\mathbb{N}_{V\uplus S}(\!|u_z\langle y\rangle.[y \leftrightarrow a]|\!)\{b/u_z\} \; !b(c).[\![L]\!]_c) \\
&= (\nu b)((\nu y)\mathbb{N}_{V'\uplus S}(\!|u_z\langle y\rangle.[y \leftrightarrow a]|\!)\{b/u_z\} \; !b(c).[\![L]\!]_c) \\
&\Rightarrow_{!\tau} (\nu b, y)(\mathbb{N}_{V'\uplus S}(\!|[y \leftrightarrow a] \mid [\![L]\!]_y \; !b(c).[\![L]\!]_c|\!)) = Q
\end{aligned}$$

Notice that $M = N[L/z] = \mathbb{E}_\Sigma(\!|z|\!)[L/z] \multimap_{\mathtt{ls}} \mathbb{E}_\Sigma(\!|L|\!)[L/z]$. Also, $Q \Rightarrow_r (\nu b)(\mathbb{N}_{V'\uplus S}(\!|[\![L]\!]_a \; !b(c).[\![L]\!]_c|\!)) = (\nu b)(\mathbb{N}_{V'\uplus S}(\!|[\![L]\!]_a\{b/u_z\} \; !b(c).[\![L]\!]_c|\!)) = [\![L[L/z]]\!]_a$, since $u_z \notin var([\![L]\!]_a)$. The result follows by taking $N = \mathbb{E}_\Sigma(\!|L|\!)[L/z]$.

□

Remark 2 *Notice that we could obtain a 1-to-1 correspondence if we worked modulo renaming of names (modulo forwarding of names), say, via a relation $\equiv_r = \Leftrightarrow_r^*$.*

5 Conclusion and Ongoing Work

We have presented a type-directed encoding from the linear substitution λ-calculus (λ_{lsub}) to a fragment of synchronous π-calculus, combining the works in [13] and [1]. We have considered the linear weak head reduction semantics for the call-by-name evaluation of the λ_{lsub} and also a π-calculus at a distance. This expands the current theoretical investigations about the expressiveness of the π-calculus via encodings from the function-as-process paradigm. An ongoing work is the proposal of an encoding from λ_{lsub}-calculus to asynchronous π-calculus based on the translation proposed in [7] to session typed processes, where the 3-1 relation for simulation may be improved, while typed relations would still play a central role. Also, it would be interesting to develop encodings to strongly normalising λ-terms and to explore its connection with intersection types.

References

[1] B. Accattoli. Evaluating functions as processes. In *TERMGRAPH*, volume 110 of *EPTCS*, pages 41–55, 2013.

[2] S. Alves, D. Nantes-Sobrinho, J. A. Pérez, and D. Ventura. An investigation of linear substitution λ-calculus assession-typed processes. Technical report, 2020.

[3] H. P. Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.

[4] G. Boudol. The π-calculus in direct style. *High. Order Symb. Comput.*, 11(2):177–208, 1998.

[5] G. Boudol and C. Laneve. Lambda-calculus, multiplicities, and the pi-calculus. In *Proof, Language, and Interaction*, pages 659–690. The MIT Press, 2000.

[6] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In P. Gastin and F. Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010.

[7] H. DeYoung, L. Caires, F. Pfenning, and B. Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In P. Cégielski and A. Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*, pages 228–242. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

[8] D. Kesner and D. Ventura. Quantitative types for the linear substitution calculus. In *IFIP TCS*, volume 8705 of *Lecture Notes in Computer Science*, pages 296–310. Springer, 2014.

[9] R. Milner. Functions as processes. *Math. Struct. Comput. Sci.*, 2(2):119–141, 1992.

[10] D. Sangiorgi. An investigation into functions as processes. In S. D. Brookes, M. G. Main, A. Melton, M. W. Mislove, and D. A. Schmidt, editors, *Mathematical Foundations of Programming Semantics, 9th International Conference, New Orleans, LA, USA, April 7-10, 1993, Proceedings*, volume 802 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 1993.

[11] D. Sangiorgi. Lazy functions and mobile processes. Technical report, RR-2515, INRIA, 1995.

[12] D. Sangiorgi and X. Xu. Trees from functions as processes. *Logical Methods in Computer Science*, 14(3), 2018.

[13] B. Toninho, L. Caires, and F. Pfenning. Functions as session-typed processes. In *FoSSaCS*, volume 7213 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2012.

Formalization of Cryptographic Algorithms in the Lean Theorem Prover

Guilherme Gomes Felix da Silva[1]

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
Rio de Janeiro, Brazil

Edward Hermann Haeusler[2]

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
Rio de Janeiro, Brazil

Bruno Lopes[3]

Instituto de Computação
Universidade Federal Fluminense
Niterói, Brazil

Abstract

Cryptographic algorithms are used in any computational resource that needs client communications. From email accounts to internet banks, the Internet relies on these algorithms to protect privacy and money. Among many others, the Data Encryption Standard (DES), the Rivest, Shamir, and Adleman public-key algorithm (RSA), and the Blowfish algorithm are widely used. Proof assistants are powerful tools designed to certify semi-automatically proofs and Lean is a new but trending one. This paper presents the certification of the soundness of these three algorithms. We describe the algorithms followed by their soundness proofs with their counterpart in Lean.

Keywords: Cryptography, Lean, Proof-assitants

1 Introduction

Cryptography is a field of computer science in which it is of the utmost importance that algorithms and protocols work properly and provide users with the expected security. These algorithms are usually tailored to encrypt a portion of data that should not be decrypted by an unauthorised user.

Usual cryptographic algorithms tend to be complex enough that a mathematical proof of correctness for them can be both difficult to understand and difficult to check for potential errors. In order to lead to the automatic verification of as-most-as-possible proof steps and ensure the usage of some kind of standard formal language, proof-assistants [3] are powerful tools strongly supported by logical frameworks. They are able to check and automatize proof steps, and reason about formalized objects.

[1] Email: guilhermegfsilva@gmail.com
[2] Email: hermann@inf.puc-rio.br
[3] Email: bruno@ic.uff.br

Among many other, The Lean Theorem Prover [4] [1] is a trending proof-assistant that can also be used as a programming language. Lean can be run directly on the web, so that the user does not need to install it and its dependencies.

The purpose of this work is to describe the usage of the Lean proof-assistant for verifying the correctness of cryptographic algorithms. The three algorithms analysed in this project are: the Data Encryption Standard (DES), the Rivest, Shamir, and Adleman public-key algorithm (RSA), and the Blowfish algorithm.

A description of the Lean Theorem Prover and its usage is introduced in Section 2. The used cryptographic algorithms are described in Section 3 with a reader-friendly description of each proof, which is then followed by the translation of said proof to Lean. Section 4 presents the conclusions and future work.

2 The Lean Theorem Prover

Launched in 2013, the Lean Theorem Prover project raised an interactive theorem prover which allows users to describe proofs via a compilable computer language. The proof assistant compiler analyzes the proof and points out any errors in it. If no errors are found, it can be assumed that the proof is correct and that the described proposition is true.

Lean is capable of modularization, so once a theorem has been proven true in Lean, it can be reused as a building block for constructing proofs of more complex theorems. It is a powerful mechanism for understanding complex proofs.

The language used for describing proofs in Lean is primarily based on types.

```
variable A : Type
variable a : A
```

What we are doing here is defining types. Lean's language interprets commands like the ones above as stating that the element to the left of the colon is a type which is contained in the one to the right of the colon. `Type` is the broadest type category in Lean, encompassing all other types, as well as variables. In the command lines above, we are defining a new type `A`, then proceeding to define a variable `a` belonging to type `A`. In this way, we can also define the premises that we will use to construct our proofs.

```
premise p1 : a = b
premise p2 : b = c
```

The equalities shown to the right of the colon operator are also interpreted by Lean as types, and objects belonging to these types are interpreted as proofs of these equalities.

To build verifiable proofs of theorems, Lean enables applying functions to existing variables and premises, thus constructing new objects which can be interpreted as new proofs.

```
theorem t1 : b = a := eq.symm p1
theorem t2 : a = c := eq.trans p1 p2
```

Using the `theorem` label, we define the proof that we want to reach after the `:` operator, and tell the user how to reach that proof after the `:=` operator. Note that it is still the user's responsibility to know how the proof itself is reached and translate said proof to the program's language; the proof assistant checks it for errors.

Consider the following example.

```
theorem t3 : c = a := eq.trans p1 p2
```

Lean would detect an error, as the type that is obtained by applying transitivity to the true premises is different from the type representing the proof that we want to reach according to the theorem header, i.e., `c = a`. However, a proof of this type could be obtained by applying the transitivity and symmetry properties in succession. This is an example of how the proof assistant distinguishes correct proofs from incorrect ones.

A description of the used commands in this work will be carried out during the proofs explanation.

3 A formalization of cryptographic algorithms in Lean

In this work we present a description of three representative cryptographic algorithms in Lean: DES, RSA, and Blowfish. DES and Blowfish are symmetric-key algorithms while RSA is a public-key algorithm which utilizes the problem of factorization of large primes as well as modular arithmetic. An overview of each algorithm and its proof of correctness is given in this section, accompanied by samples of code showing how these proofs were translated into Lean's language. Due to lack of space we cannot present here the full description in Lean, but it is available at https://github.com/GGFSilva/LeanCryptographyProof.

[4] https://leanprover.github.io

3.1 *Data Encryption Standard (DES)*

The Data Encryption Standard [2] (DES) is a symmetric-key algorithm approved by the NBS in 1976. DES was the primary algorithm used in most symmetric-key systems until it was largely replaced by the Advanced Encryption Standard (AES). It uses a 56-bit key and encrypts 64 bits of data at a time, breaking larger blocks of data into 64-bit blocks and encoding them individually.

A data encoding is done by applying an encryption step 16 times in succession to the 64-bit data block. Let L_i and R_i, respectively, the leftmost 32 bits and the rightmost 32 bits of the data block after i encryption steps such that $L_{i+1} = R_i$, $R_{i+1} = L_i \oplus f(R_i, k_i)$, where f is a sub-function of DES which performs bit substitution via table lookup, and k_i is a 48-bit subkey derived from a main key k. This function encodes only the right half of the data block, then swaps it with the left half so that the latter will be encoded in the next step. In addition to these 16 encryption steps, a round of DES encryption runs two bit permutation operations, one before the main encryption steps and one after. To decode a data block, the process is simply to run the algorithm again over the encoded data block, but with the 16 subkeys $k_i, 1 \leq i \leq 16$ in reverse order.

We have formalized both operations in Lean and proved that the decrypted block always corresponds to the original one. So, we aimed to prove that $L_1^d = R_{15}$ and that $R_1^d = L_{15}$ where the superscript d stands for the decrypted version.

So, from the algorithm definition we assume the following premises.

```
premise H1        : L16_E = R15_E
premise H2        : R16_E = xor L15_E (f R15_E K_16)
premise H3        : L0_D  = R16_E
premise H4        : R0_D  = L16_E
premise H5        : L1_D  = R0_D
premise H6        : R1_D  = xor L0_D (f R0_D K_16)
```

The proof is carried out as follows.

```
theorem ProofRight_Long : R15_E = L1_D :=
have H7 : R15_E = L16_E, from eq.symm H1,
have H8 : L16_E = R0_D, from eq.symm H4,
have H9 : R0_D = L1_D, from eq.symm H5,
have H10 : R15_E = R0_D, from eq.trans H7 H8
  ,
show R15_E = L1_D, from eq.trans H10 H9
```

Header of the theorem
Proof of $L_{15} = R_{16}$ by application of symmetry to H1
Proof of $L_{16} = R_1^d$ by application of symmetry to H4
Proof of $R_0^d = L_1^d$ by application of symmetry to H5
Proof of $R_{15} = R_0^d$ by application of transitivity to H7 and H8
Proof of $R_{15} = L_1^d$ by application of transitivity to H10 and H9

Then we prove that $R_1^d = L_{15}$.

```
theorem ProofLeft : L15_E = R1_D :=
have H11 : R1_D = xor L0_D  (f R0_D  K_16),
    from H6,
have H12 : R1_D = xor R16_E (f R0_D  K_16),
    from eq.subst H3 H11,
have H13 : R1_D = xor R16_E (f L16_E K_16),
    from eq.subst H4 H12,
have H14 : R1_D = xor R16_E (f R15_E K_16),
    from eq.subst H1 H13,
have H15 : R1_D = xor (xor L15_E (f R15_E
    K_16)) (f R15_E K_16),
    from eq.subst H2 H14,
have H16 : xor (xor L15_E (f R15_E K_16)) (f
      R15_E K_16) =
            xor L15_E (xor (f R15_E K_16) (f
                R15_E K_16)),
    from xor_assoc L15_E (f R15_E K_16) (f
        R15_E K_16),
have H17 : R1_D = xor L15_E (xor (f R15_E
    K_16) (f R15_E K_16)),
    from eq.trans H15 H16,
have H18 : xor L15_E (xor (f R15_E K_16) (f
    R15_E K_16)) = L15_E,
    from xor_cancel L15_E (f R15_E K_16),
show L15_E = R1_D, from eq.symm (eq.trans
    H17 H18)
```

Header of the theorem
Proof of $R_1^d = L_0^d \oplus f(R_0^d, k_{16}))$, from H6
Proof of $R_1^d = R_{16} \oplus f(R_0^d, k_{16}))$, by substituting H3 in H11
Proof of $R_1^d = R_{16} \oplus f(L_{16}, k_{16}))$, by substituting H4 in H12
Proof of $R_1^d = R_{16} \oplus f(R_{15}, k_{16}))$, by substituting H1 in H13
Proof of $R_1^d = (L_{15} \oplus f(R_{15}, k_{16})) \oplus f(R_{15}, k_{16}))$, by substituting H2 in H14
Proof of $(L_{15} \oplus f(R_{15}, k_{16})) \oplus f(R_{15}, k_{16})) = L_{15} \oplus (f(R_{15}, k_{16}) \oplus f(R_{15}, k_{16}))$, by associativity of $\oplus$
Proof of $R_1^d = L_{15} \oplus (f(R_{15}, k_{16}) \oplus f(R_{15}, k_{16}))$ by application of transitivity to H15 e H16
Proof of $L_{15} \oplus (f(R_{15}, k_{16}) \oplus f(R_{15}, k_{16})) = L_{15}$ by cancellation of $\oplus$
Proof of $L_{15} = R_1^d$ by application of transitivity to H17 and H18 and application of symmetry to the result

3.2 RSA

RSA [4] is a public-key cryptography algorithm developed by Ron Rivest, Adi Shamir, and Leonard Adleman. Being a public-key algorithm (also known as an asymmetric-key algorithm) means each run of the algorithm by a user requires a pair of keys (a public key and a private key). The public key, which is used for encoding messages, is shared to other users so they can freely encode messages for the user who generated the keys; this user can then proceed to decode the messages with his private key. As the public key cannot decode messages, there is no harm in divulging it. Therein lies the purpose of public-key algorithms: to solve the problem of how to divulge a key without compromising security.

RSA is based on the one-way function derived from the problem of factorization of large primes: for two sufficiently large primes p and q, it is relatively easy to determine $n = p \times q$ but computationally infeasible in reasonable time to obtain p and q from n. Public and private key pair is determined as follows.

(i) Two sufficiently large primes are chosen such that $n = p \times q$
(ii) Euler's Totient function is computed such that $\phi(n) = (p-1) \times (q-1)$
(iii) An exponent d is chosen such that d and $\phi(n)$ shares no natural divisor; so there exists e such that $d \times e \equiv 1 (\text{mod } \phi(n))$[5] (this property will be proven), composing (n, d) and n, e as private and public keys respectively
(iv) The encoding and decoding function are, respectively, $y = x^e \text{ mod } n$ and $x = y^d \text{ mod } n$

The fact that RSA works properly can be proven by showing that the decoding function retrieves the original data. It is equivalent to prove that $x = (x^e \text{ mod } n)^d \text{ mod } n$. This proof is split in two cases[6].

(i) gcd$(x, n) = 1$

$$x^{\phi(n)} \equiv 1(\text{mod } n) \quad (1)$$
$$\left(x^{\phi(n)}\right)^t \equiv 1^t(\text{mod } n) \quad (2)$$
$$\left(x^{\phi(n)}\right)^t \equiv 1(\text{mod } n) \quad (3)$$
$$x \times (x^{t\times\phi(n)}) \equiv x(\text{mod } n) \quad (4)$$
$$(x^{1+t\times\phi(n)} \equiv x(\text{mod } n) \quad (5)$$
$$x^{d\times e} \equiv x(\text{mod } n) \quad (6)$$

(ii) gcd$(x, n) \neq 1$

Let $x = r \times p$ for some $r \in \mathbb{Z}$ and $\gcd(x, q) = 1$.

$$x^{\phi(q)} \equiv 1(\text{mod } q) \quad (7)$$
$$\left(x^{\phi(q)}\right)^t \equiv 1^t(\text{mod } q) \quad (8)$$
$$\left(x^{\phi(q)}\right)^t \equiv 1(\text{mod } q) \quad (9)$$
$$\left(\left(x^{\phi(q)}\right)^t\right)^{(p-1)} \equiv 1^{(p-1)}(\text{mod } q) \quad (10)$$
$$\left(\left(x^{\phi(q)}\right)^{(p-1)}\right)^t \equiv 1(\text{mod } q) \quad (11)$$
$$\left(x^{(q-1)\times(p-1)}\right)^t \equiv 1(\text{mod } q) \quad (12)$$
$$\left(x^{\phi(n)}\right)^t \equiv 1(\text{mod } q) \quad (13)$$
$$\left(x^{\phi(n)}\right)^t \equiv 1 + (v \times q) \quad (14)$$
$$x \times \left(x^{\phi(n)}\right)^t \equiv x + (x \times v \times q) \quad (15)$$
$$x \times \left(x^{\phi(n)}\right)^t \equiv x + (r \times p \times v \times q) \quad (16)$$
$$x \times \left(x^{\phi(n)}\right)^t \equiv x + (n \times r \times v) \quad (17)$$
$$x \times \left(x^{\phi(n)}\right)^t \equiv x(\text{mod } n) \quad (18)$$

As it is possible to reach $x^{d\times e} \equiv x(\text{mod } n)$ from $x \times \left(x^{\phi(n)}\right)^t \equiv x(\text{mod } n)$, then the correctness proof is complete. To formalize this proof in Lean it was required to implement several auxiliary functions and attribute properties. Due to lack of space we omit parts of the formalization. The headers of gcd, and ϕ are presented below. Also, two auxiliary functions to determine if some number is prime and whether two numbers are congruent in a given modulus.

[5] We remember that in modular arithmetics $a \equiv b(\text{mod } m)$ iff $\exists n \in \mathbb{Z}$ such that $a = b + n \times m$.

[6] By *gcd* we denote the greatest common divisor.

```
variable gcd (a b : nat) : nat
variable phi (n   : nat) : nat
variable prime     (a            : nat) : Prop
variable congruent (a b modulus : nat) : Prop
```

The premises adopted are the following.

```
premise nDef    : n = p * q
premise pPhi    : phi p = p - one
premise qPhi    : phi q = q - one
premise nPhi    : phi n = (q - one) * (p - one)
premise xLess   : x < n
premise pIsPrime : prime p
premise qIsPrime : prime q
premise de_Inverse : (congruent (d * e) one (phi n))
premise ModuloDef (a b n : nat) :
congruent a b n equiv exists c, a = b + (c * n)
premise CongruenceReflexivity (a b n : nat) :
congruent a b n implies congruent b a n
premise CongruenceScaling (a b n k : nat) :
congruent a b n implies congruent (a * k) (b * k) n
premise CongruenceExponentiation (a b n k : nat) :
congruent a b n implies congruent (a and k) (b and k) n
premise OneMul : forall n : nat, one * n = n
premise OneExp : forall n : nat, one and n = one
premise ExpOne : forall n : nat, n and one = n
premise ExpSum  (a b c : nat) : (a and b) * (a and c) = a and (b + c)
premise ExpMul  (a b c : nat) : (a and b) and c = a and (b * c)
premise ExpSwap (a b c : nat) : (a and b) and c = (a and c) and b
premise EqMul (a b k : nat) :
a = b implies k * a = k * b
premise MulDistrib (a b c : nat) :
a * (b + c) = (a * b) + (a * c)
premise Euler (a b : nat) :
gcd a b = one implies congruent (a and (phi b)) one b
premise GCDProperty (a b c : nat) :
gcd a (b * c) neq one implies prime b implies prime c implies a < b * c implies
(((exists n, a = n * b) and (gcd a c = one)) or
 ((exists n, a = n * c) and (gcd a b = one))
```

$n = p \times q$, by definition
$\phi(p) = p - 1$
$\phi(q) = q - 1$
$\phi(n) = (q-1) \times (p-1)$

$x < n$, by definition
p is prime
q is prime
$d \times e \equiv 1(\text{mod } \phi(n))$
$a \equiv b(\text{mod } n)$ iff $\exists c, a = b + (c \times n)$

If $a \equiv b(\text{mod } n)$ then $a \times k \equiv (b \times k)(\text{mod } n)$

If $a \equiv b(\text{mod } n)$ then $a^k \equiv b^k(\text{mod } n)$

For any natural number n, $1 \times n = n$

For any natural number n, $1^n = 1$
For any natural number n, $n^1 = n$

For any natural numbers a, b and c, $a^b \times a^c = a^{b+c}$

For any natural numbers a, b and c, $\left(a^b\right)^c = a^{b \times c}$

For any natural numbers a, b and c, $\left(a^b\right)^c = (a^c)^b$

If $a = b$ then $k \times a = k \times b$

$a \times (b + c) = (a \times b) + (a \times c)$

Euler's theorem

If $\gcd(a, b) \neq 1$, b and c are prime and $a < b \times c$ then there exists n such that either $a = n \times b$ and $\text{mdc}(a, c) = 1$ or $a = n \times c$ and $\text{mdc}(a, b) = 1$

Due to the lack of space we present only the `theorem` declarations for (i) and (ii) and the final steps. The full proof is available at https://github.com/GGFSilva/LeanCryptographyProof.

```
theorem ProofCoprime (t : nat) :
gcd x n = one implies congruent (x * ((x and (phi n)) and t)) x n :=
```

finishing with

```
show congruent (x * ((x and (phi n)) and t)) x n,
from eq.subst (mul.comm ((x and (phi n)) and t) x) H6
```

```
theorem ProofNotCoprime_Part1 :
gcd x n neq one implies
(((exists n, x = n * p) and (gcd x q = one)) or
 ((exists n, x = n * q) and (gcd x p = one))
) :=

show (((exists n, x = n * p) and (gcd x q = one)) or
       ((exists n, x = n * q) and (gcd x p = one))),
    from GCDProperty x p q H3 pIsPrime qIsPrime H2

ProofNotCoprime_Part2 (t : nat) :
(exists n, x = n * p) and (gcd x q = one) implies
congruent (x * ((x and (phi n)) and t)) x n :=

show congruent (x * ((x and (phi n)) and t)) x n,
    from (iff.elim_right
        (ModuloDef (x * ((x and (phi n)) and t)) x n)) this))

theorem ProofFinal (t : nat) :
congruent (x * ((x and (phi n)) and t)) x n implies
congruent (x and (one + ((phi n) * t))) x n :=

show congruent (x and (one + ((phi n) * t))) x n,
    from eq.subst (ExpSum x one ((phi n) * t)) H3
```

3.3 Blowfish

Blowfish [5] is a symmetric-key algorithm which encodes 64 bits of data at a time via 16 iterations of an encoding function. Unlike DES, however, Blowfish alters all 64 bits of data in each of the 16 iterations and uses a total of 18 keys. The algorithm encodes data with the functions $L_{i+1} = R_i \oplus F(L_i \oplus P_{i+1})$, $R_{i+1} = L_i \oplus P_{i+1}$, $L_{16} = (R_{15} \oplus F(L_{15} \oplus P_{16}) \oplus P_{17})$ and $R_{16} = (L_{15} \oplus P_{16}) + P_{18}$.

We show that for some data encrypted, after it be decrypted we recover the original data. It is equivalent to show that $L_0 R_0 = L^d_{16} R^d_{16}$. This proof is divided in ten parts (five for left and five for the right).

Due to the lack of space we point the reader to https://github.com/GGFSilva/LeanCryptographyProof for the correctness proof and its formalization in the Lean Theorem Prover.

4 Conclusions

This paper introduced the correctness proofs and their implementation in the Lean Theorem Prover three classical and representative cryptographic algorithms: DES, RSA and Blowfish. All of the proofs are available at https://github.com/GGFSilva/LeanCryptographyProof and may be checked.

Further work includes the implementation in other proof-assistants (such as Coq[7] and/or Isabelle[8]) for future inclusion in the Logipedia[9], an open online encyclopaedia of proofs.

References

[1] de Moura, L., S. Kong, J. Avigad, F. van Doorn and J. von Raumer, *The lean theorem prover*, in: *25th International Conference on Automated Deduction (CADE-25)*, 20015.

[2] Diffie, W. and M. Hellman, *Exhaustive cryptanalysis of the NBS Data Encryption Standard*, IEEE Computer **10** (1977), pp. 74–84.

[3] Loveland, D. W., "Automated Theorem Proving: a logical basis," Elsevier, 2014.

[4] Rivest, R., A. Shamir and L. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM **21** (1978), pp. 120–126.

[5] Schneier, B., *Description of a new variable-length key, 64-bit block cipher (Blowfish)*, in: *Cambridge Security Workshop*, 1993.

[7] https://coq.inria.fr

[8] https://isabelle.in.tum.de

[9] http://logipedia.inria.fr

Agda formalization of a security-preserving translation from flow-sensitive to flow-insensitive security types

Cecilia Manzino[1]

Departamento de Ciencias de la Computación
Universidad Nacional de Rosario
Argentina

Alberto Pardo[2]

Instituto de Computación
Universidad de la República
Montevideo, Uruguay

Abstract

The analysis of information flow is a popular technique for ensuring the confidentiality of data. It is in this context that confidentiality policies arise for giving guarantees that private data cannot be inferred by the inspection of public data. One of those policies is non-interference, a semantic condition that ensures the absence of illicit information flow during program execution by not allowing to distinguish the results of two computations when they only vary in their confidential inputs. A remarkable feature of non-interference is that it can be enforced statically by the definition of information flow type systems. In those type systems, if a program type-checks, then it means that it meets the security policy.
In this paper we focus on the preservation of non-interference through program translation. Concretely, we formalize the proof of security preservation of Hunt and Sands' translation that transforms high-level While programs typable in a flow-sensitive type system into equivalent high-level programs typable in a flow-insensitive type system. Our formalization is performed in the dependently-typed language Agda. We use the expressive power of Agda's type system to encode the security type systems at the type level. A particular aspect of our formalization is that it follows a fully internalist approach where we decorate the type of the abstract syntax with security type information in order to obtain the representation of well-typed (i.e secure) programs. A benefit of this approach is that it allows us to directly express the property of security preservation in the type of the translation relation. In this manner, apart from inherently expressing the transformation of programs, the translation relation also stands for an inductive proof of security preservation.

Keywords: non-interference, information flow type systems, dependently-typed programming, Agda, type safety

1 Introduction

The analysis of information flow is a popular technique for ensuring the confidentiality of data. It is in this context that confidentiality policies arise for giving guarantees that private data cannot be inferred by the inspection of public data. Non-interference [3] is an example of a security policy. It is a semantic condition that ensures the absence of illicit information flow during program execution by not allowing to distinguish the results of two computations when they only vary in their confidential inputs. A remarkable feature of non-interference is that it can be enforced statically by the definition of an information flow type system [2,16,14,8]. Thus, when a program type-checks in such a type system then it means that it satisfies the security policy. In this setting, program variables are classified in different categories (types) according with the kind of information they can store (e.g., public or confidential data). The advantage of modelling security properties

[1] Email: `ceciliam@fceia.unr.edu.ar`

[2] Email: `pardo@fing.edu.uy`

in terms of types is that they can be checked at compile-time, thus partially reducing or even eliminating the overhead of checking properties at run-time.

Most of the security type systems are *flow-insensitive* [14]. These are type systems in which the security level of the program variables remain unchanged. This contrasts with security type systems that are *flow-sensitive* [4,13]. In those type systems each variable can have a different security level at different points of the program. Flow-sensitive type systems are more permissive than flow-insensitive ones since they accept a larger set of secure programs.

In this paper we focus on the preservation of non-interference through program translation. Concretely, we formalize the proof of security preservation of Hunt and Sands' translation [4] that transforms high-level While programs typable in a flow-sensitive type system into equivalent high-level programs typable in a flow-insensitive type system. Our formalization is performed in the dependently-typed functional language Agda [9,1]. We use the expressive power of Agda's type system to encode the security type systems at the type level.

A particular aspect of our formalization is that it follows a fully internalist approach [10,7,12], where we decorate the type of the abstract syntax with security type information in order to obtain the representation of well-typed (i.e secure) programs. These are terms that simulaneously represent ASTs and (their associated) typing rules in the formal type system [15,11]. An interesting consequence of this representation is that it restricts the object language terms that are representable. Indeed, not every AST is representable, but only those that are well-typed according to the type system of the object language. But even more interesting is the effect that this representation has on functions between typed terms: only functions that preserve those type invariants are accepted. The positive aspect of this fact is that its verification reduces to type-checking.

The paper is organized as follows. In Section 2 we present the high-level language that serves as source and target of the translation and a flow-insensitive type system for it. Section 3 deals with the flow-sensitive type system for the language defined by Hunt and Sands. Section 4 presents the program transformation and the proof that it preserves security typing. Section 5 concludes the paper.

The complete Agda code is available at `https://www.fceia.unr.edu.ar/~ceciliam/codes/`.

2 Type-insensitivity

We start with a summary description of the language that serves as source and target of the translation. After that we present one of the type systems that enforces secure information flow for programs of the language. The system to be shown in this section is a flow-insensitive type system and corresponds to the system that is used to type the target programs of the translation. We decided to start by describing the endpoint of the translation because the system is more natural and easier to understand. It is also a suitable context where to introduce the internalist approach we want to pursue for the Agda implementation.

Out of differences in implementation details, the flow-insensitive type system to be shown in this section has already been presented in [7], where we developed a Haskell implementation of a security-preserving compiler also using an internalist approach.

2.1 The language

The language to be used in the translation is a standard While language with expressions and statements defined by the following abstract syntax:

$$\begin{aligned} e &::= n \mid x \mid e_1 + e_2 \\ S &::= x := e \mid \texttt{skip} \mid S_1;S_2 \mid \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ e\ \texttt{do}\ S \end{aligned}$$

where $e \in \mathbf{Exp}$ (expressions), $S \in \mathbf{Stm}$ (statements), $x \in \mathbf{Var}$ (variables) and $n \in \mathbf{Num}$ (integer literals).

The semantics is completely standard. The meaning of both expressions and statements is given relative to a state $s \in \mathbf{State} = \mathbf{Var} \to \mathbf{Num}$, which contains the current value of each variable.

We assume the semantics for expressions is given by an evaluation function $\mathcal{E} : \mathbf{Exp} \to \mathbf{State} \to \mathbf{Num}$ defined by induction on the structure of expressions. For statements, we define a big-step semantics whose transition relation is written as $\langle S, s\rangle \Downarrow s'$, meaning that the evaluation of a statement S in an initial state s terminates with a final state s'. The definition of the transition relation is presented in Figure 1.

Notice that, for simplicity, the language does not contain boolean expressions. Instead, the condition of an `if` or a `while` statement is given by an arithmetic expression such that the condition is true when the expression evaluates to zero, and false otherwise.

2.2 Security Type System

Suppose we want to model a security scenario where each program variable has associated a security level stating the degree of confidentiality of the values it stores. In such a context it is natural to implement some

$$\frac{}{\langle x := e,\, s\rangle \Downarrow s[x \mapsto \mathcal{E}[\![e]\!]\, s]} \qquad \frac{}{\langle \texttt{skip},\, s\rangle \Downarrow s} \qquad \frac{\langle S_1,\, s\rangle \Downarrow s' \quad \langle S_2,\, s'\rangle \Downarrow s''}{\langle S_1;S_2,\, s\rangle \Downarrow s''}$$

$$\frac{\mathcal{E}[\![e]\!]\, s = 0 \quad \langle S_1,\, s\rangle \Downarrow s'}{\langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\, s\rangle \Downarrow s'} \qquad \frac{\mathcal{E}[\![e]\!]\, s \neq 0 \quad \langle S_2,\, s\rangle \Downarrow s'}{\langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\, s\rangle \Downarrow s'}$$

$$\frac{\mathcal{E}[\![e]\!]\, s = 0 \quad \langle S,\, s\rangle \Downarrow s' \quad \langle \texttt{while}\ e\ \texttt{do}\ S,\, s'\rangle \Downarrow s''}{\langle \texttt{while}\ e\ \texttt{do}\ S,\, s\rangle \Downarrow s''} \qquad \frac{\mathcal{E}[\![e]\!]\, s \neq 0}{\langle \texttt{while}\ e\ \texttt{do}\ S,\, s\rangle \Downarrow s}$$

Figure 1. Big-step semantics of statements

security mechanism in order to protect confidential data. We will do so by implementing an information flow type system [16,14,8]. The type system to be defined in this section is *flow-insensitive* in the sense that it considers that the security level of the variables is maintained unchanged during program execution. Variables with this property are called fixed-type variables. This contrasts with the so-called floating-type variables, whose security level may vary along program excecution. We will deal with floating-type variables in the *flow-sensitive* type system to be presented in Section 3.

We assume a bounded lattice of security levels $(\mathcal{L}, \leq)$ with meet $(\wedge)$ and join $(\vee)$ operations, and top $(\top)$ and bottom $(\bot)$ values. The bottom value represents the least security level (*public* data) whereas the top value represents the highest security level (*confidential* data). We also assume that the lattice comes with an equality relation $\approx$ between levels. An expression $l < l'$ means that security level l is less confidential than l'.

Non-interference is a property on programs that guarantees the absence of illicit information flows during their execution. An illicit flow occurs when information flows from variables of higher security level to variables with lower security level. A program satisfies this security property when the final value of any variable with security level l is not influenced by a variation of the initial value of variables with higher security level. This property can be formulated in terms of program semantics. We write x_l to refer to a variable with security level l. Let us say that two states s and s' are l-equivalent, written $s \cong_l s'$, when every variable with lower security level than l contains the same value in both states; i.e. $s(x_{l'}) = s'(x_{l'})$ for every $x_{l'}$ with $l' < l$. The significance of l-equivalence is that l-equivalent states are indistinguishable to a lower than l confidential observer (i.e. an observer that can only inspect data with security level less than l).

A program $S \in \textbf{Stm}$ is said to be **non-interfering** when, for any pair of l-equivalent initial states, if the execution of S starting on each of these states terminates, then it does so in l-equivalent final states:

$$\textbf{NI}(S) \overset{\mathrm{df}}{=} \forall s_i, s'_i.\ s_i \cong_l s'_i \ \wedge\ \langle S,\, s_i\rangle \Downarrow s_f \ \wedge\ \langle S,\, s'_i\rangle \Downarrow s'_f \implies s_f \cong_l s'_f$$

This definition of non-interference is *termination-insensitive* in the sense that it does not take into account non-terminating executions of programs.

It is well-known that this property can be checked statically by the definition of an information-flow type system that enforces noninterference [16,14]. In Figure 2 we present a syntax-directed security type system for our language (alternative formulations for the type system can be found in [8,6]). Security levels are used as types and are referred to as security types. The reason for presenting a syntax-directed type system is because it is the appropriate formulation to be considered later for the implementation.

Expressions The type system for expressions uses a judgement of the form $\vdash e : st$, where $st \in \mathcal{L}$. According to this system, the security type of an expression is the maximum of the security types of its variables. Integer numerals are considered public data.

Statements The goal of secure typing for statements is to prevent improper information flows during program execution. Information flow can appear in two forms: explicit or implicit.

An *explicit flow* is observed when data are copied to less confidential variables. Consider two variables x_H and y_L, with $L < H$. For example, the assignment $y_L := x_H + 1$ is not allowed because the value of the variable x_H is copied to a less confidential variable, y_L. On the other hand, an assignment in the opposite direction, e.g. $x_H := y_L$, is authorized, since it does not represent a security violation.

Implicit information flows arise from the control structure of the program. The following is an example of an insecure program where an implicit flow occurs (again assume $L < H$):

$$\texttt{if}\ x_H\ \texttt{then}\ y_L := 1\ \texttt{else skip}$$

The reason for being insecure is because by observing the value of the variable y_L on different executions we can infer information about the value of the variable x_H. This is because we are performing the assignment of a variable with security type L in a more confidential context (in this case, the branch of a conditional

EXPRESSIONS

$$\vdash n : \bot \qquad \vdash x_t : t \qquad \frac{\vdash e : st \quad \vdash e' : st'}{\vdash e + e' : st \vee st'}$$

STATEMENTS

$$[pc] \vdash \texttt{skip} \ \textsc{skip} \qquad \frac{\vdash e : st \quad st \leq t \quad pc \leq t}{[pc] \vdash x_t := e}\ \textsc{ass} \qquad \frac{[pc] \vdash S_1 \quad [pc] \vdash S_2}{[pc] \vdash S_1 ; S_2}\ \textsc{seq}$$

$$\frac{\vdash e : st \quad [st \vee pc] \vdash S_1 \quad [st \vee pc] \vdash S_2}{[pc] \vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2}\ \textsc{if} \qquad \frac{\vdash e : st \quad [st \vee pc] \vdash S}{[pc] \vdash \texttt{while}\ e\ \texttt{do}\ S}\ \textsc{while}$$

Figure 2. Flow-insensitive Security Type System

statement with a condition of type H). Due to these situations it is necessary to keep track of the security level of the program counter in order to know the security level of the context in which a sentence occurs. On the other hand, a program like this:

$$\texttt{if}\ y_L + 2\ \texttt{then}\ z_L := y_L + 1\ \texttt{else}\ x_H := x_H - 1$$

is accepted because the final value of the variable z_L only depends on the initial value of y_L.

The typing judgement for statements has the form $[pc] \vdash S$ and means that S is typable in the security context pc. Rule ASS states that an assigment to a variable x_t can be done in a context lower or equal than t; explicit flows are prevented by the restriction $st \leq t$. In order to prevent implicit flows, the rules IF and WHILE impose a restriction between the security level of the condition and the branches of the conditional or the body of the while. In a context pc, if the condition has type st, then the branches (of the `if`) or the body (of the `while`) must type in a context which is the least upper bound of pc and st.

A desirable property for a security type system is type soundness, which means that every typable statement satisfies non-interference. We build up the soundness proof using two lemmas taken from [8]. The first one, called *confinement*, states that if a sentence is typable in a context pc then the execution of the sentence does not alter the value of the variables with level lower than pc.

Lemma 2.1 (CONFINEMENT) $[pc] \vdash S \wedge \langle S, s\rangle \Downarrow s' \implies s \cong_l s'$, *with* $l < pc$.

Proof The proof can be done trivially by induction on the derivations of the evaluation relation $\langle S, s\rangle \Downarrow s'$.

The second lemma, called *anti-monotonicity*, states that if a sentence is typable in a context pc, then is also typable in any context lower than pc.

Lemma 2.2 $[pc] \vdash S \wedge pc' \leq pc \implies [pc'] \vdash S$.

Proof Straightforward by induction on the derivation of $[pc] \vdash S$.

Based on these lemmas, we can now state type soundness.

Theorem 2.3 (TYPE SOUNDNESS) $[pc] \vdash S \implies \mathbf{NI}(S)$.

We formalized the proof of this theorem in Agda for a lattice of two levels; the proof is available at `https://www.fceia.unr.edu.ar/~ceciliam/codes/proofs`.

2.3 Implementation

Now we present an Agda implementation of the abstract syntax and the type system. We proceed following an internalist approach where we attach type invariants (in our case typing information) to our abstract syntax representations. Proceeding that way we obtain the respresentation of *well-typed* expressions and statements.

We represent the expressions by means of a type family Exp, which is indexed by a value of type 𝕊 representing the security type of the expression. 𝕊 is the carrier set of our lattice of security types.

```
data Exp   : 𝕊 → Set where
   IntVal  : ℕ → Exp ⊥
```

$$\textsc{Skip}\ \frac{}{pc \vdash \Gamma\ \{\ \texttt{skip}\ \}\ \Gamma} \qquad \textsc{Assign}\ \frac{\Gamma \vdash e\ :\ t}{pc \vdash \Gamma\ \{\ x := e\ \}\ \Gamma[x \mapsto pc \vee t]}$$

$$\textsc{Seq}\ \frac{pc \vdash \Gamma\ \{\ S_1\ \}\ \Gamma' \quad pc \vdash \Gamma'\ \{\ S_2\ \}\ \Gamma''}{pc \vdash \Gamma\ \{\ S_1;S_2\ \}\ \Gamma''} \qquad \textsc{If}\ \frac{\Gamma \vdash e\ :\ t \quad pc \vee t \vdash \Gamma\ \{\ S_i\ \}\ \Gamma'_i \quad i = 1,2}{pc \vdash \Gamma\ \{\ \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \}\ \Gamma'_1 \sqcup \Gamma'_2}$$

$$\textsc{While}\ \frac{\Gamma'_i \vdash e\ :\ t_i \quad pc \vee t_i \vdash \Gamma'_i\ \{\ S\ \}\ \Gamma''_i \quad 0 \leq i \leq n}{pc \vdash \Gamma\ \{\ \texttt{while}\ e\ \texttt{do}\ S\ \}\ \Gamma'_n} \qquad \Gamma'_0 = \Gamma,\ \Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma,\ \Gamma'_{n+1} = \Gamma'_n$$

Figure 3. Flow-sensitive Type System

```
  Var      : (x : Fin n) → (st : 𝕊) → Exp st
  Add      : Exp st → Exp st' → Exp (st ∨ st')
```

The internalist reprentation of statements is given by the following type family, which is indexed by a security type representing the security level of the program context in which the statement is executed.

```
data Stm   : 𝕊 → Set where
  Skip     : Stm   pc
  Assign   : (x : Fin n) → (y : 𝕊) → st ≼ y → pc ≼ y → Exp st → Stm pc
  Seq      : Stm pc → Stm pc → Stm pc
  If0      : Exp st → Stm (pc ∨ st) → Stm (pc ∨ st) → Stm pc
  While    : Exp st → Stm (pc ∨ st) → Stm pc
```

Notice that the constructors of Exp and Stm are a direct implementation of the typing rules in Figure 2. A benefit of the internalist approach is that now, the typing judgement S : Stm pc directly corresponds to the judgement $[pc] \vdash S$ in our formal type system.

For reasons that will be clear later on when we see the flow-sensitive type system, instead of simply using naturals, we use elements from a finite set (Fin n) to represent variables. As a consequence of this, our representation of expressions and statements in parameterised by a natural n.

3 Flow-Sensitive Type System

In a flow-sensitive type system, the security type of a variable can change during program execution. This allows us to type more secure programs than with a flow-insensitive type system. For example, consider the following code, again with $L < H$,

$$x_L := y_H;\ x_L := 0$$

Although in the second assignment the variable x_L is overridden with the constant 0, this code is rejected by a flow-insensitive type system because it has an insecure statement ($x_L := y_H$). This is, however, accepted by a flow-sensitive type system because the security level of x_L is relabeled to H after the first assignment.

Another example of an intuitively secure code that is rejected by a flow-insensitive type system is the following:

$$y_H := 0;\ \textbf{if}\ y_H\ \textbf{then}\ x_L := 1\ \textbf{else}\ x_L := 2$$

This code is secure since the value of the variable y_H is overwritten with 0 before the conditional and therefore by inspecting the value of x_L we cannot know anything about the value of x_H before assigning it to 0.

3.1 Type System

Now we present the flow-sensitive type system that was defined by Hunt and Sands [4,5] for the While language. This is the type system that is used to type-check the source programs of the translation. Like before, the type system is parameterized by a lattice of security types $\mathcal{L}$.

Concerning the expressions, the typing judgement $\Gamma \vdash e\ :\ t$ now requires a type environment Γ because the type system deals with floating-type variables. Again, the type of an expression is defined as the least upper bound of the types of its variables.

For statements, the judgement is now of the form $pc \vdash \Gamma\ \{\ S\ \}\ \Gamma'$, meaning that S is typable in the security context pc and the type environments Γ and Γ'. Those environments (called pre- and post-environment,

$$\Gamma'_0 = \Gamma, \Gamma''_0 = \Gamma[w \mapsto N][z \mapsto M] \qquad \Gamma \vdash w > 0 \,:\, L \qquad \frac{L \vdash \Gamma \;\{\; w := y \;\}\; \Gamma[w \mapsto N] \quad L \vdash \Gamma[w \mapsto N] \;\{\; z := x + 1 \;\}\; \Gamma''_0}{L \vdash \Gamma \;\{\; w := y; z := x + 1 \;\}\; \Gamma''_0}$$

$$\Gamma'_1 = \Gamma''_0 \sqcup \Gamma = \Gamma[w \mapsto N] \qquad \Gamma'_1 \vdash w > 0 \,:\, N \qquad \frac{N \vdash \Gamma'_1 \;\{\; w := y \;\}\; \Gamma'_1 \quad N \vdash \Gamma'_1 \;\{\; z := x + 1 \;\}\; \Gamma'_1[z \mapsto H]}{N \vdash \Gamma'_1 \;\{\; w := y; z := x + 1 \;\}\; \Gamma'_1[z \mapsto H]}$$

$$\Gamma'_2 = \Gamma''_1 \sqcup \Gamma = \Gamma[w \mapsto N] = \Gamma'_1$$

Figure 4. Example of a typing iteration

respectively) describe the security level of the variables before and after the execution of S. A post-environment Γ' is now necessary because the security level of the variables may be changed by the execution of a statement.

Figure 3 shows a syntax-directed flow-sensitive type system for statements. The formulation of the type system has an algorithmic character in the sense that $pc \vdash \Gamma \;\{S\}\; \Gamma'$ computes the least post-environment Γ' obtained after the execution of statement S in a pre-environment Γ and context pc. Like in the case of flow-insensitivity, having a syntax-directed formulation of the type system results helpful because it allows us to represent it in terms of type families.

Environments form a join-semilattice whose structure is inherited from that of the lattice of security levels. In this sense, the join of two environments ($\sqcup$) is defined pointwise: $\Gamma \sqcup \Gamma' = \lambda x. \Gamma(x) \vee \Gamma'(x)$; the same with the partial order between two environments: $\Gamma \sqsubseteq \Gamma'$ iff $\forall x. \Gamma(x) \leq \Gamma'(x)$. The semilattice has a top element given by the environment with all variables in the top security type ($\top$).

According to rule Assign, after an assignment the type of a variable x may: (i) change to a higher value $pc \vee t$ if the assignment is performed in a context pc and the assigned expression is of type t, with $\Gamma(x) < pc \vee t$; (ii) change to a lower value if $\Gamma(x) > pc \vee t$; or (iii) remain unaltered otherwise.

As usual, the rules for If and While are designed to prevent implicit flows. The branches (of the conditional) or the body (of the while) must be typable in a context which is the least upper bound of the context pc and the type t of the condition.

For example, considering the following lattice of security types:

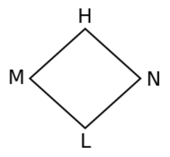


Figure 4 shows the steps of the typing iteration for the code: `while` $x > 0$ `do` $w := y; z := x + 1$ starting with the environment $\Gamma = \{w : \mathsf{L}, x : \mathsf{M}, y : \mathsf{N}, z : \mathsf{H}\}$.

Notice that in the While rule the post-environment is the result of an iterative construction which iterates until a fixed-point is obtained. The body of the loop is repeatedly typed until the post-environment does not change with respect to the last iteration. The rule can then be reformulated in terms of a least fixed point operator:

$$\textsc{While-fix}\; \frac{\Gamma_f = \mathrm{fix}(\lambda\Gamma \;.\; \text{let} \;\; \Gamma \vdash e \,:\, t \quad pc \vee t \vdash \Gamma \;\{\; S \;\}\; \Gamma' \;\text{in}\; \Gamma' \sqcup \Gamma_0)}{pc \vdash \Gamma_0 \;\{\; \texttt{while}\; e \;\texttt{do}\; S \;\}\; \Gamma_f}$$

This fixed point construction is guaranteed to terminate because it is computed on a monotone function (defined over the typing rules) and the set of environments is finite. The proof of convergence of this rule was given in [4] as part of the proof of the following theorem, which states the correctness of the type system. We refer as $\mathcal{A}^S$ to the function that calculates the least Γ' such that $pc \vdash \Gamma \;\{\; S \;\}\; \Gamma'$.

Theorem 3.1 ([4]) *For all S, pc, Γ, there exists a unique Γ' such that $pc \vdash \Gamma \;\{\; S \;\}\; \Gamma'$ and furthermore, the corresponding function $\mathcal{A}^S(pc, \Gamma) \mapsto \Gamma'$ is monotone.*

To implement this type system in Agda we need to define a function that computes the fixed point of the While-fix rule. The construction of the fixed point is based on the Agda formalization of the following theorem (Theorem 3.2), which states the existence of a fixed point for any monotone function that satisfies certain requirements on an arbitrary partially ordered set.

In the formulation of the theorem we capture the relevant characteristics that our semilattice of environments has. Taking into account that environments are finite mappings that contain only the variables that occur in the analysed program, it turns out that the semilattice is finite. From that, what is relevant for us is that every strictly ascending chain in the semilattice has finite length; in particular, every chain that ends at the top environment. This will be reflected in the theorem by assuming the existence of a function, called *bound*, that associates a natural number to every element of the poset and is strictly decreasing with respect to the strict order:

$$x \sqsubset y \Rightarrow \mathit{bound}\, y < \mathit{bound}\, x.$$

The intuition behind the value of *bound* is that it represents the length of the longest strictly ascending chain from an element to top ($\top$). The value of *bound* at $\top$ is of course zero. Instead of requiring that the poset has a top element and that the value of *bound* at $\top$ is zero, we will equivalently require that *bound* has a unique minimal element:

$$\mathit{bound}\ x = 0\ \wedge \mathit{bound}\ y = 0 \Rightarrow x = y.$$

Theorem 3.2 *Let $(S, \sqsubseteq)$ be a poset, $x \in S$ an element, $g : S \to S$ a monotone function over $\sqsubseteq$, and bound : $S \to \mathbb{N}$ a strictly decreasing function with respect to the strict order ($\sqsubset$) and with a unique minimal element. If there is an element $x \in S$ such that $x \sqsubseteq g\ x$ and bound $x \leq k$ for some k, then there exists $n \leq k$ such that $g^n\ x$ is a fixed-point of g.*

Proof The proof is by induction on k.

- $k = 0$: By hypothesis *bound* $x \leq 0$, and therefore *bound* $x = 0$. Since *bound* is a decreasing function and $x \sqsubseteq g\ x$, we have that *bound* $(g\ x) \leq$ *bound* x. Thus, *bound* $(g\ x) = 0$. By uniqueness of *bound*'s minimal element we conclude that $g\ x = x$, and therefore x is a fixed-point of g.
- $k = k' + 1$: By hypothesis we know that *bound* $x \leq k' + 1$ and $x \sqsubseteq g\ x$. Then we have two cases:
  case $x = g\ x$ **:** Then x is a fixed-point of g.
  case $x \sqsubset g\ x$ **:** Since *bound* is strictly decreasing wrt $\sqsubset$ we have that *bound* $(g\ x) <$ *bound* $x\ \leq k' + 1$, and therefore *bound* $(g\ x) \leq k'$. Since g is monotone and $x \sqsubset g\ x$ we have that $g\ x \sqsubseteq g^2\ x$. Then, by induction hyphotesis we conclude that there exists $n \leq k'$ such $g\ ((g^n\ (g\ x)) = g^n\ (g\ x)$, meaning that $g^n\ (g\ x)$ is a fixed-point of g. Therefore, we conclude that $g^{n+1}\ x$ is a fixed-point of g where $n + 1 \leq k' + 1$. □

Based on this theorem we implement in Agda a function fixS that computes the fixpoint. 𝕊 denotes the carrier set of the poset, ≤ its order relation, ≈ the equality in 𝕊 (which we assume is decidable). We write ≤ℕ to the denote less or equal between natural numbers; ♯ denotes the bound function which we assume is strictly decreasing wrt the strict order in the poset.

```
fixS :  -- boundary of iterations
        (k : ℕ)
        -- monotone function
        (g : 𝕊 → 𝕊) →
        (∀ {x y} → x ≤ y → g x ≤ g y) →
        -- initial value
        (s : 𝕊) →
        s ≤ g s →
        -- invariant
        ♯ s ≤ℕ k →
        Σ 𝕊 (λ x → x ≈ g x )
```

Function fixS turns out to be itself a monotone function wrt the elements of 𝕊. This fact should not surprise because we are simply working with an abstraction of the fixpoint construction we had in the type system for statements. Later we will use the monotonicity of fixS for the implementation of the type system.

3.2 Implementation

For the Agda implementation of the language with a flow-sensitive type system we want to proceed in a similar way as we did for the flow-insensitive type system using an internalist approach. Again, the goal is to define type families that represent (flow-sensitive) typed terms for expressions and statements. On those typed terms we will define later the translation to flow-insensitive typed terms.

For expressions the internalist implementation is immediate. Expressions are represented by a type ExpS, which is parametrized by the type environment (given by a vector of security types), and the security type of the expression. Like before, 𝕊 denotes the carrier set of the lattice of security types.

```
data ExpS     (Γ : Vec 𝕊 n) : 𝕊 → Set where
   IntValS   : ℕ → ExpS Γ ⊥
   VarS      : (x : Fin n) → ExpS Γ (lookup x Γ )
```

```
    AddS    : ExpS Γ st → ExpS Γ st' → ExpS Γ (st ∨ st')
```

Variable n denotes the number of program variables that occur in the program code. Each program variable is identified by a value in the finite set Fin n. That way, each variable has associated a position in a type environment of type Vec $\mathbb{S}$ n.

The internalist implementation of statements is in terms of a type family called StmS, which is indexed by the program counter and the (pre- and post-) environments. A judgement $pc \vdash \Gamma\ \{S\}\ \Gamma'$ is then represented by a typing judgement in Agda: S : StmS Γ pc Γ'. Like in Section 2.3, this is indeed possible because we have a syntax-directed type system.

The implementation of the flow-sensitive typing rules for skip, assigments, conditionals and sequences is direct and poses no difficulty. For while statements, however, the situation is other. The process is a bit more laborious because we have to deal with the computation of the fixpoint. Such computation requires that we iterate over the typing judgement corresponding to the body of the while and that is impossible to be done with a decorated term of type StmS. This leads us to implement the fixpoint construction on ordinary undecorated abstract syntax terms, an undesirable situation that seems to be unavoidable. The only terms that will be represented in that undecorated form will be the term that represents the body of the while and the term corresponding to the loop condition. This will require to implement the typing relation for statements separately (in a classical externalist fashion) in order to be able to type check the body of the loop in each iteration.

We start with the definition of the undecorated abstract syntax and the relation TyStm, which implements the typing rules for statements in an externalist fashion. This requires the definition of a fix function in order to implement the While-fix rule. As second step we define the decorated type family StmS.

The undecorated abstract syntax for expressions and statements is given by the following type families. Both datatypes are indexed by the number of variables in the program code.

```
  data ASTExp (m : ℕ) : Set where
    INTVAL : ℕ → ASTExp m
    VAR    : Fin m → ASTExp m
    ADD    : ASTExp m → ASTExp m → ASTExp m

  data ASTStm (m : ℕ) : Set where
    ASSIGN : Fin m → ASTExp m → ASTStm m
    IF0     : ASTExp m → ASTStm m → ASTStm m → ASTStm m
    WHILE   : ASTExp m → ASTStm m → ASTStm m
    SEQ     : ASTStm m → ASTStm m → ASTStm m
```

The type system for statements is implemented by the following relation. Function tyExp computes the security type of an expression under an environment Γ.

```
  data TyStm : ASTStm n → 𝕊 → Vec 𝕊 n → Vec 𝕊 n → Set where
    Skip : {Γ : Vec 𝕊 n}{ pc : 𝕊} → TyStm SKIP pc Γ Γ
    Ass : {x : Fin n}{e : ASTExp n}{Γ : Vec 𝕊 n}{pc : 𝕊} →
      TyStm (ASSIGN x e) pc Γ (change x Γ (pc ∨ (tyExp Γ e)))
    Seq : {Γ Γ' Γ" : Vec 𝕊 n}{pc : 𝕊} {s₁ s₂ : ASTStm n} →
      TyStm s₁ pc Γ Γ' →
      TyStm s₂ pc Γ' Γ" →
      TyStm (SEQ s₁ s₂) pc Γ Γ"
    If0 : {Γ Γ' Γ" : Vec 𝕊 n}{pc : 𝕊}{e : ASTExp n}{s₁ s₂ : ASTStm n} →
      TyStm s₁ (pc ∨ (tyExp Γ e)) Γ Γ' →
      TyStm s₂ (pc ∨ (tyExp Γ e)) Γ Γ" →
      TyStm (IF0 e s₁ s₂) pc Γ (Γ' ⊔ Γ")
    While : {Γ : Vec 𝕊 n}{pc : 𝕊}{e : ASTExp n}{s : ASTStm n} →
      TyStm (WHILE e s) pc Γ (proj₁ (fix s e pc Γ))
```

Given an initial environment Γ_0 and a program context pc, we know that the post-environment that the rule for while computes is the result of the following fixpoint construction:

$$\text{fix}(\lambda\Gamma\ .\ \text{let}\ \ \Gamma \vdash e : t \quad pc \sqcup t \vdash \Gamma\ \{\ S\ \}\ \Gamma'\ \text{in}\ \Gamma' \sqcup \Gamma_0)$$

We implement the computation of this fixpoint by a function called fix with the following type:

```
fix : {n : ℕ} → (s : ASTStm n) → (e : ASTExp n) → (pc : 𝕊) → (Γ₀ : Vec 𝕊 n) →
   Σ (Vec 𝕊 n) (λ Γ → Γ ≈≈ body e s pc Γ₀ Γ)
```

where ≈≈ is the equality between type environments. Function fix is defined in terms of fixS, where function body plays the role of function g. Function body implements the body of the fixpoint construction:

```
body :   ASTExp n → -- the condition
          ASTStm n → -- while's body
         𝕊   →    -- program counter's security level
         Vec 𝕊 n → -- initial environment Γ
         Vec 𝕊 n → -- environment Γ'ᵢ
         Vec 𝕊 n   -- computed environment Γ'ᵢ₊₁

body e s pc Γ Γ' =    let    st = tyExp Γ' e
                              Γ" = proj₁ (tyStm s (pc ∨ st) Γ')
                       in (Γ" ⊔ Γ)
```

Function body is defined in terms of function tyStm, a functional implementation of the type system, which computes a post-environment and a proof that it is indeed the environment that results from the type system.

```
tyStm : (s : ASTStm n) → (pc : 𝕊) → (Γ : Vec 𝕊 n) → Σ (Vec 𝕊 n) (λ Γ' → TyStm s pc Γ Γ')
tyStm SKIP pc Γ = Γ , Skip
tyStm (ASSIGN x e) pc Γ = change x Γ (pc ∨ (tyExp Γ e)) , Ass
tyStm (SEQ s s') pc Γ =    let (Γ' , tcs)    = tyStm s pc Γ
                                (Γ" , tcs') = tyStm s' pc Γ'
                            in    Γ" , Seq tcs tcs'
tyStm (IF0 e s s') pc Γ =   let pc' = pc ∨ (tyExp Γ e)
                                (Γ' , tcs)    = tyStm s pc' Γ
                                (Γ" , tcs') = tyStm s' pc' Γ
                            in    Γ' ⊔ Γ" , If0 tcs tcs'
tyStm {n} (WHILE e s) pc Γ = proj₁ (fix s e pc Γ) , While
```

The partial order relation between environments (_⊑_) is given by the pointwise ordering between the corresponding vectors:

```
data _⊑_ : Vec 𝕊 n → Vec 𝕊 n → Set ℓ₂ where
  ⊑-nil    : [] ⊑ []
  ⊑-cons : ∀ {st st' : 𝕊} {Γ Γ' : Vec 𝕊 n} → Γ ⊑ Γ' → st ≤ st' → (st :: Γ) ⊑ (st' :: Γ')
```

Together with the join operation (⊔), the partial order of environments (⊑) turns out to form a join semilattice.

Function body turns out to be monotone. Its monotonocity requires the monotonicity of tyStm, which in turn requires that fix is monotone. In fact, body, tyStm and fix, so as their monotonicity proofs are mutually recursive.

```
bodyMonotone : {e : ASTExp n}{s : ASTStm n} {pc pc' : 𝕊}{Γ Γ' Γ₁ Γ₁' : Vec 𝕊 n} →
   pc ≤St pc' → Γ ⊑ Γ₁ → Γ' ⊑ Γ₁' → body e s pc Γ Γ' ⊑ body e s pc' Γ₁ Γ₁'

tcMonotone : {pc pc' : 𝕊}{Γ Γ₁ : Vec 𝕊 n}
   (s : ASTStm n) →
   pc ≤St pc' → Γ ⊑ Γ₁ → tyStm s pc Γ ⊑ tyStm s pc' Γ₁

fixMonotone : {pc pc' : 𝕊}{Γ Γ' : Vec 𝕊 n} →
   (s : ASTStm n) → (e : ASTExp n) →
   pc ≤St pc' → Γ ⊑ Γ' → proj₁ (fix s e pc Γ) ⊑ proj₁ (fix s e pc' Γ')
```

In order to define fix in terms of fixS we need that the semilattice of environments posesses a *bound* function. We use function sumDist for that end. Given an environment, sumDist returns the sum of the distances to the top type (⊤) that have the types of the variables in the environment. For that we need to assume that the lattice of security types also has associated a bound function that we call ♯.

```
sumDist : {n : ℕ} → Vec 𝕊 n → ℕ
sumDist [] = 0
sumDist (st :: Γ) = ♯ st + sumDist Γ
```

sumDist fits well as a bound function because in each step of the fixpoint computation the security type of each variable in the computed environment is possibly higher. Therefore, if at least one of the security types change to a higher type then the global distance to the top type decreases. This function also has a unique minimal value because only the top environment (that with all variables with the highest security type) has global distance equal zero.

```
sumDistDecr : {Γ Γ' : Vec 𝕊 n} → Γ ⊏ Γ' → sumDist Γ' <ℕ sumDist Γ

minimalEnv : {Γ Γ' : Vec 𝕊 n} → sumDist Γ ≡ 0 → sumDist Γ' ≡ 0 → Γ ≡ Γ'
```

Finally, we have all the elements to define fix.

```
fix :   (s : ASTStm n) →
         (e : ASTExp n) →
        (pc : 𝕊) →
        (Γ : Vec 𝕊 n) →
        Σ (Vec 𝕊 n) (λ Γ' → Γ' ≈≈ body e s pc Γ Γ')
fix s e pc Γ =
   let   Γ₀ = Γ
         Γ₁ = body e s pc Γ Γ₀
   in fixS   (n * ♯ ⊥)
              (body e s pc Γ) -- function g
             (bodyMonotone {e = e} {s = s} refl≤ (refl⊑ {Γ = Γ}))
             Γ₀
             Γ⊑Γ'⊔Γ
             (initInv Γ₀)
```

where function initInv is the proof that for any environment Γ, sumDist $\Gamma \leq n$ * ♯ ⊥. This corresponds to the initial state of the invariant that fixS requires to the iteration boundary, which is initialized in n * ♯ ⊥ (the global distance to the top of the bottom environment).

The decorated type StmS is indexed by a program context and the pre- and post- environment. The constructors SkipS, AssignS, SeqS and IfS are direct implementations of the typing rules. This contrasts to the WhileS constructor, which needs to deal with undecorated ASTs to compute the fixpoint.

```
data StmS : Vec 𝕊 n → 𝕊 → Vec 𝕊 n → Set c where

AssignS :   {Γ : Vec 𝕊 n}{pc st : 𝕊} →
              (x : Fin n) →
            ExpS Γ st →
            StmS Γ pc (change x Γ (pc ∨ st))
SkipS     :   {Γ : Vec 𝕊 n}{pc : 𝕊} →
               StmS Γ pc Γ
SeqS      :   {Γ Γ' Γ" : Vec 𝕊 n}{pc : 𝕊} →
               StmS Γ pc Γ' →
              StmS Γ' pc Γ" →
              StmS Γ pc Γ"
IfS       :   {pc st : 𝕊}{Γ Γ' Γ" : Vec 𝕊 n} →
               ExpS Γ st →
              StmS Γ (pc ∨ st) Γ' →
              StmS Γ (pc ∨ st) Γ" →
              StmS Γ pc (Γ' ⊔ Γ")
WhileS    :   {Γ : Vec 𝕊 n}{pc : 𝕊}
```

$$\frac{}{pc \vdash \Gamma\ \{\texttt{skip} \leadsto \texttt{skip}\}\ \Gamma} \qquad \frac{\Gamma \vdash E : t \qquad s = pc \vee t}{pc \vdash \Gamma\ \{x := E \leadsto x_s := E_\Gamma\}\ \Gamma[x \mapsto s]}$$

$$\frac{pc \vdash \Gamma\ \{S_1 \leadsto D_1\}\ \Gamma' \qquad pc \vdash \Gamma'\ \{S_2 \leadsto D_2\}\ \Gamma''}{pc \vdash \Gamma\ \{S_1;S_2 \leadsto D_1;D_2\}\ \Gamma''}$$

$$\frac{\Gamma \vdash E : t \qquad pc \vee t \vdash \Gamma\ \{S_1 \leadsto D_1\}\ \Gamma'_1 \qquad pc \vee t \vdash \Gamma\ \{S_2 \leadsto D_2\}\ \Gamma'_2 \qquad \Gamma' = \Gamma'_1 \sqcup \Gamma'_2}{pc \vdash \Gamma\ \{\texttt{if}\ E\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \leadsto \texttt{if}\ E_\Gamma\ \texttt{then}\ (D_1\ ;\ \Gamma' := \Gamma'_1)\ \texttt{else}\ (D_2\ ;\ \Gamma' := \Gamma'_2)\}\ \Gamma'}$$

$$\frac{\Gamma'_i \vdash E : t_i \qquad pc \vee t_i \vdash \Gamma'_i\ \{S \leadsto D_i\}\ \Gamma''_i \qquad 0 \leq i \leq n}{pc \vdash \Gamma\ \{\texttt{while}\ E\ \texttt{do}\ S \leadsto \Gamma'_n := \Gamma\ ;\texttt{while}\ E_{\Gamma'_n}\ \texttt{do}\ (D_n\ ;\ \Gamma''_n := \Gamma'_n)\}\ \Gamma'_n}$$
$$\Gamma'_0 = \Gamma, \Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma, \Gamma'_{n+1} = \Gamma'_n$$

Figure 5. Translation rules

```
 (e : ASTExp n) →
(s : ASTStm n ) →
StmS Γ pc (proj₁ (fix s e pc Γ))
```

Finally, the following lifting function summarizes the internalization process.

```
liftS : {Γ Γ' : Vec 𝕊 n}{pc : 𝕊} (s : ASTStm n) → TyStm s pc Γ Γ' → StmS Γ pc Γ'
liftS SKIP           Skip        = SkipS
liftS (ASSIGN x e)   Ass         = AssignS x (liftE e)
liftS (SEQ s₁ s₂)    (Seq t₁ t₂) = SeqS (liftS s₁ t₁) (liftS s₂ t₂)
liftS (IF0 e s₁ s₂)  (If0 t₁ t₂) = IfS   (liftE e) (liftS s₁ t₁) (liftS s₂ t₂)
liftS (WHILE e s)    While       = WhileS e s
```

4 Translation to flow-insensitive

Now we turn to the formalization of Hunt and Sands's translation in Agda. It converts programs typable in the flow-sensitive type system to equivalent programs typable in the flow-insensitive type system. The transformation rules, shown in Figure 5, are defined as an extension to the flow-sensitive type system and are expressed in terms of the judgement $pc \vdash \Gamma\ \{S \leadsto D\}\ \Gamma'$, where as before pc is the security context, and Γ and Γ' are type environments. S is a statement with floating-type variables, which is typable in the flow-sensitive type system; D is an equivalent statement (or statements) produced by the program translation, with fixed-type variables and typable in the flow-insensitive type system.

To transform a program typable in the flow-sensitive system to another typable in the flow-insensitive system we need to transform floating-type variables into fixed-type variables. The set of fixed-type variables of a program, $FVar$, is obtained from the set of floating variables, Var, by annotating each variable name with a security type:

$$FVar = \{x_t \mid x \in Var,\ t \in \mathcal{L}\}$$

Each time a floating-type variable x raises its security type from t to s, with $t < s$, the translation will reflect this fact by constructing an assignment that moves information from x_t to x_s. Therefore, the transformed code may include a sequence of variable assignments (between annotated variables) that make explicit the variables that changed their security level. We write $\Gamma := \Gamma'$ to represent an appropriate sequentialisation of the following set of variable assignments:

$$\{x_s := x_t \mid \Gamma(x) = s, \Gamma'(x) = t, s \neq t\}$$

A sequence of assignments $\Gamma := \Gamma'$ is used with environments Γ, Γ' such that $\Gamma' \sqsubseteq \Gamma$. This gives rise to well-typed assignments as information flows in the appropriate direction. The security context in which a sequence of assigments must type is the minimun security type of the assigned variables. We define a funtion min that calculates that context:

```
min: (Γ : Vec 𝕊 n ) → (Γ' : Vec 𝕊 n ) → 𝕊
min [] [] = ⊤
min (st :: Γ) (st' :: Γ') with st ≟ st'
```

```
... | yes st≈st' = min Γ Γ'
... | no st≉st' = st ∧ (min Γ Γ')
```

where $\overset{?}{=}$ denotes the decidability of equality in the set $\mathbb{S}$ of security types. A sequence of assignments has then the following type:

```
Γ:=Γ' : (Γ : Vec 𝕊 n) → (Γ' : Vec 𝕊 n) → Γ' ⊑ Γ → Stm ( min Γ Γ' )
```

This function generates a sequence of assignments by traversing the environments Γ and Γ'. For each i it adds an assigment $\Gamma(i) := \Gamma'(i)$ if $\Gamma(i) \sqsupset \Gamma'(i)$.

Let us now see the translation function. Given an environment Γ, every expression E with floating-type variables can be transformed to an expression E_Γ with fixed-type variables by replacing each floating-type variable x by a fixed-type variable x_s, where $s = \Gamma(x)$. The following function implements the transformation, where Exp is the type of expressions presented in Section 2.

```
transExp : { st : 𝕊}{Γ : Vec 𝕊 n} → ExpS Γ st → Exp st
transExp (IntValS y) = IntVal y
transExp {Γ = Γ} (VarS x) = Var x (lookup x Γ)
transExp (AddS y y') = Add (transExp y) (transExp y')
```

The translation for statements follows the rules shown in Figure 5:

```
translate : {pc : 𝕊} {Γ Γ' : Vec 𝕊 n} → StmS Γ pc Γ' → Stm pc
translate {pc = pc} {Γ = Γ } (AssignS {st = st} x e) =
  let x≤x∨y , x≤y∨x , _ = supremum pc st
  in Assign x (pc ∨ st) x≤y∨x   x≤x∨y (transExp e)
translate SkipS = Skip

translate (SeqS s s') = Seq (translate s) (translate s')

translate (IfS {pc} {st} {Γ} {Γ₁'} {Γ₂'} e s s') =
  let   ass₁ = Γ:=Γ' (Γ₁' ⊔ Γ₂') Γ₁' Γ⊑Γ⊔Γ'
        ass₂ = Γ:=Γ' (Γ₁' ⊔ Γ₂') Γ₂' Γ⊑Γ'⊔Γ
        p≼m : (pc ∨ st) ≼ min (Γ₁' ⊔ Γ₂') Γ₁'
        p≼m = propMin' s s'
        ass₁' = lemmaA2A {pc' = pc ∨ st} ass₁ p≼m   -- Γ' := Γ₁'
        p≼m₂ : (pc ∨ st) ≼ min (Γ₁' ⊔ Γ₂') Γ₂'
        p≼m₂ = propMin'' s s'
        ass₂' = lemmaA2A {pc' = pc ∨ st} ass₂ p≼m₂   -- Γ' := Γ₂'
  in If0 (transExp e) (Seq (translate s) ass₁') (Seq (translate s') ass₂')

translate (WhileS {Γ} {pc}   e s)
  =  let  Γn'      =  proj₁ (fix s e pc Γ)
          stn' , eΓn'  =  fromAstE Γn' e
          Γn'' , prf   =  tyStm s (pc ∨ stn') Γn'
          dn       =  translateASTStm   (pc ∨ stn') Γn' s
          ass      =  lemmaA2A   {pc' = pc}   -------------- Γn' := Γ
                                  (Γ:=Γ' Γn' Γ (Γ⊑Γn' {Γ = Γ} e s ))
                                 (lemmaA2C {pc = pc}
                                 (WhileS {Γ = Γ} {pc = pc} e s))

          ass₂     =  lemmaA2A   {pc' = pc ∨ stn'} -------------- Γn' := Γn''
                                  (Γ:=Γ' Γn'' Γn' (Γn'⊑Γn''  e s) )
                                 (lemmaA2C   {pc = pc ∨ stn'} (liftS s prf) )
       in Seq ass (While (transExp eΓn') (Seq dn ass₂))
```

Function translateASTStm is like translate, but it works on undecorated ASTs for statements:

```
translateASTStm : (pc : A) (Γ : Vec A n) → ASTStm n → Stm pc
```

The functions lemmaA2A and lemmaA2C are the formalization of the lemmas given in the paper [4] as part of proof of the static soundness property for the translation. Here we show just the type of these functions:

```
lemmaA2A : { pc pc' : 𝕊} → Stm pc → pc' ≼ pc →Stm pc'

lemmaA2C : {pc : 𝕊}{Γ Γ' : Vec 𝕊 n} → StmS Γ pc Γ' → pc ≼ (min Γ' Γ )
```

It is worth noticing that our translate not only implements the desired translation but it also ensures that the statement that results from the translation is typable in the flow-insensitive type system. That is, we are using Agda's type system to enforce the preservation of non-interference during translation. In other words, translate can be understood as an implementation of the following theorem, where $[pc] \vdash D$ is the typing judgement for statements presented in Section 2.

Theorem 4.1 ([4]) *If* $pc \vdash \Gamma \, \{S \rightsquigarrow D\} \, \Gamma'$ *then* $[pc] \vdash D$*.*

5 Conclusions

We presented the formalization of Hunt and Sands's translation for high-level secure programs. translation uses a type-based approach to noninterference to converts programs in a While language, typable in a flow-sensitive type system, into programs in the same language typable in a flow-insensitive type system. Agda was our formalization framework; we used it both as a functional language and as a proof assistant.

In both versions of the language we introduced an internalist, typed representations of the abstract syntax both for expressions and statements. In the target version of the language, the internalist representation was the result of a direct implementation of the typing rules. In the source version of the language, however, it was necessary to combine the internalist representation with an externalist one in order to deal with the computation of a fixpoint.

A distinguishing feature of our Agda formalization was the systematic use we did of typed representations of the abstract syntax for the two versions of the object language. This was possible thanks to the syntax-directed formulations of the security type systems and the translation relation. As a result of this encoding, only terms corresponding to non-interfering programs can be written in the Agda implementations of the language. This has also consequences on the functions we can define between those typed representations. In particular, this happens with the implementation of Hunt and Sands's translation. In our formalization we defined the translation as a function that preserves well-typed representations of abstract syntax terms. The equations of the translation function then correspond to the proof steps of the preservation property. The verification that the proof is correct is then performed by Agda's type-checker.

References

[1] Bove, A. and P. Dybjer, *Dependent types at work*, in: A. Bove, L. S. Barbosa, A. Pardo and J. S. Pinto, editors, *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, Lecture Notes in Computer Science **5520** (2008), pp. 57–99.
URL https://doi.org/10.1007/978-3-642-03153-3_2

[2] Denning, D. E., *A lattice model of secure information flow*, Commun. ACM **19** (1976), pp. 236–243.
URL http://doi.acm.org/10.1145/360051.360056

[3] Goguen, J. A. and J. Meseguer, *Security policies and security models*, in: *Symposium on Security and Privacy* (1982), pp. 11–20.

[4] Hunt, S. and D. Sands, *On flow-sensitive security types*, SIGPLAN Not. **41** (2006), pp. 79–90.
URL http://doi.acm.org/10.1145/1111320.1111045

[5] Hunt, S. and D. Sands, *From exponential to polynomial-time security typing via principal types*, in: *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, 2011, pp. 297–316.
URL https://doi.org/10.1007/978-3-642-19718-5_16

[6] Manzino, C., "Security preserving program translations," Master's thesis, PEDECIBA Informática, Universidad de la República, Uruguay (2018).

[7] Manzino, C. and A. Pardo, *A Security Types Preserving Compiler in Haskell*, in: F. M. Q. Pereira, editor, *Proceedings of the 18th Brazilian Symposium on Programming Languages - SBLP 2014, Maceio, Brazil, October 2-3, 2014.*, Lecture Notes in Computer Science **8771** (2014), pp. 16–30.
URL https://doi.org/10.1007/978-3-319-11863-5_2

[8] Nipkow, T. and G. Klein, “Concrete Semantics: With Isabelle/HOL,” Springer Publishing Company, Incorporated, 2014.

[9] Norell, U., *Dependently typed programming in Agda*, in: *4th international workshop on Types in Language Design and Implementation*, TLDI ’09 (2009), pp. 1–2.
URL http://doi.acm.org/10.1145/1481861.1481862

[10] Pardo, A., E. Gunther, M. Pagano and M. Viera, *An internalist approach to correct-by-construction compilers*, in: D. Sabel and P. Thiemann, editors, *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018* (2018), pp. 17:1–17:12.
URL http://doi.acm.org/10.1145/3236950.3236965

[11] Pasalic, E. and N. Linger, *Meta-programming with typed object-language representations*, in: *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings*, 2004, pp. 136–167.

[12] Poulsen, C. B., A. Rouvoet, A. Tolmach, R. Krebbers and E. Visser, *Intrinsically-typed definitional interpreters for imperative languages*, Proc. ACM Program. Lang. **2** (2018), pp. 16:1–16:34.
URL https://doi.org/10.1145/3158104

[13] Russo, A. and A. Sabelfeld, *Dynamic vs. static flow-sensitive security analysis*, in: *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, 2010, pp. 186–199.
URL https://doi.org/10.1109/CSF.2010.20

[14] Sabelfeld, A. and A. C. Myers, *Language-based information-flow security*, IEEE J. Selected Areas in Communications **21** (2003), pp. 5–19.

[15] Sheard, T., *Languages of the future*, SIGPLAN Not. **39** (2004), pp. 119–132.
URL http://doi.acm.org/10.1145/1052883.1052897

[16] Volpano, D. M. and G. Smith, *A type-based approach to program security*, in: *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT ’97 (1997), pp. 607–621.
URL http://dl.acm.org/citation.cfm?id=646620.697712

Strong Normalization for the Simply-Typed Lambda Calculus in Constructive Type Theory Using Agda

Sebastián Urciuoli[1] Álvaro Tasistro Nora Szasz[2]

Universidad ORT Uruguay
Montevideo, Uruguay

Abstract

We consider a pre-existing formalization in Constructive Type Theory of the pure Lambda Calculus in its presentation in first order syntax with only one sort of names and alpha-conversion based upon multiple substitution, as well as of the system of assignment of simple types to terms. On top of it, we formalize a slick proof of strong normalization given by Joachimski and Matthes whose main lemma proceeds by complete induction on types and subordinate induction on a characterization of the strongly normalizing terms which is in turn proven sound with respect to their direct definition as the accessible part of the relation of one-step beta reduction. The proof of strong normalization itself is thereby allowed to consist just of a direct induction on the type system. The whole development has been machine-checked using the system Agda. We assess merits and drawbacks of the approach.

Keywords: Formal Metatheory, Lambda Calculus, Constructive Type Theory

1 Introduction

In [3] we have presented a formalization of the Lambda Calculus in Constructive Type Theory using first-order syntax, with only one sort of names for both bound and free variables, without identifying terms up to alpha conversion, and using multiple (simultaneous) substitution as fundamental operation.

It was then our aim to investigate whether this very concrete approach was in any way amenable to full formalization. The approach is historically rooted in the detailed meta-theoretical study of the calculus by Curry and Feys [7] where (unary) substitution was given a well-known definition as a total meta-operation on terms on top of which all the relevant relations of the calculus where defined, starting with alpha conversion. That definition of substitution proceeds by recursion on the size of terms, but its full justification requires a simultaneous proof that its use as a renaming operation (i.e. when the substituted term is a variable) does not affect the size of the term wherein it is effected. This makes its formalization in e.g. Constructive Type Theory and using any of the proof assistants by now available, extremely difficult or even impossible.

However, the use of simultaneous multiple substitution as suggested in e.g. [11] brought about the possibility of achieving a development of the meta-theory of the calculus that was at the same time fully formal –to the extent of being close to a machine-checkable version– and humanly readable. Indeed, as explained in [11,3] the use of this kind of substitution makes it possible to avoid the recursion on the size of terms since, when crossing over lambdas the bound name is changed to an appropriate one and the corresponding renaming recorded into an enlarged, multiple substitution that goes on to operate on the body of the abstraction. Hence the effect of substitution can be given a simple definition by structural recursion which makes it plausible that the meta-theory of the calculus can be pursued using also simple forms of induction. The fact that bound names are always changed permits to avoid case analyses too.

[1] Partially supported by Agencia Nacional de Investigación e Innovación (ANII), Uruguay.

[2] Email: `{urciuoli,tasistro,szasz}@ort.edu.uy`

Our work in [3,6] verified the preceding hypotheses to some extent: we were able to carry out full formalizations of the meta-theory of the pure Lambda Calculus up to the Standardization Theorem, and of the simply typed Lambda Calculus (more precisely, of the system of assignment of simple types to terms) up to the Subject-Reduction Theorem. Our main conclusion from those efforts has been that, in spite of having to explicitly consider alpha conversion, the workload was manageable, the formal machine-checked proofs not extremely lengthy, and their presentation in detailed mathematical English readable and reasonably complete. We also generated a basic Agda library [2] that was successfully employed and enlarged by a Master's student in [6].
In the present paper we wish to continue the preceding line of work, with the purposes of:

- testing the foregoing hypotheses against challenges of increasing complexity, and
- making progress towards a comprehensive corpus of fully formal meta-theory of the Lambda Calculus.

We therefore tackle the problem of formally proving Strong Normalization for the system of assignment of simple types to terms. The method of proof we choose to formalize is due to [8] where the authors present slick proofs of weak and strong normalization which proceed by simple induction on the type system. These proofs use a main lemma that allows to deal with the difficult case of applications in the induction, via a separate complete induction on types, with a subordinate induction which, in the case of strong normalization, proceeds on a characterization of the strong normalizing terms originally given in [13].
In [8], vector notation is used for presenting terms which we shall avoid for a matter of convenience of the formalization. Also, –like in most of meta-theory developments– terms are identified under alpha conversion, with the usual conventions for choice of bound names being made use of without further ado. Then, for example, variable capture is not take into account –which of course, we will have to. We wish to test whether:

- our formalization can still be kept within limits of reasonable complexity,
- our pre-existing Agda library makes the grade as to continue supporting this additional development, and
- the effort is within the reach of a newcomer with limited experience in full formalization, the use of proof assistants, and Agda in particular. Actually, the work forms part of the Master's thesis of the first author. The whole development has been formalised in Constructive Type Theory and machine-checked in the system Agda [10]. The corresponding code is available at `https://github.com/surciuoli/lambda-metatheory`.

We proceed now to the development itself. Its structure is as follows: in the next section we present the basic definitions, and results that are used in this formalization –some from [3], and some new developments. In Section 3 we give two definitions of the strongly normalizing terms. The first is the more natural one, defined as the accessible part of the one-step beta reduction, and the other is a syntactic characterization originally given (in vector notation) in [13]. Our formulation of the latter is mutual with the predicate characterizing strongly normalizing neutral terms and the strong head reduction. We formalize the proof of soundness of the second definition with respect to the original one in Section 4. In Section 5 we make use of the syntactic characterization to obtain a proof of the Strong Normalization Theorem for typable terms in the system of assignments of simple types, which ultimately consists of a direct induction on the type system. Finally, in Section 6 we present the overall conclusions.

2 Preliminaries

2.1 Syntax

We start with a denumerable type V of names, also to be called variables; i.e. for concreteness we can put $\mathsf{V} =_{def} \mathbb{N}$ or $\mathsf{V} =_{def} \mathsf{String}$. Letters x, y, z with primes or subindices shall stand for variables. Terms are defined inductively as usual —below we show a grammar for the abstract syntax:

Definition 2.1 *The terms in Λ are defined by the following grammar:*

$$M, N ::= x \mid MN \mid \lambda x M.$$

M, N, P, $Q\ldots$ range over lambda terms. In the concrete syntax we assume the usual convention according to which application binds tighter than abstraction.

Definition 2.2 *That a variable x is free in M is written $x * M$ and its opposite is written $x \# M$, to be pronounced x fresh in M as in nominal techniques. Both relations are defined inductively in a standard manner.*

2.2 Substitutions

We work with multiple (simultaneous) substitutions, i.e. mappings associating a term to every variable. Actually, the substitutions we must handle are identity almost everywhere, and so we can generate them by starting up from the empty substitution ι, which maps each variable to itself as a term, and applying the update operation defined as follows:

Definition 2.3 (Substitution Update) *$(\sigma, x{:=}M)$ is the substitution defined by:*

$$\begin{aligned}&(\sigma, x{:=}M)\ x = M\\&(\sigma, y{:=}M)\ x = \sigma x,\ \ \textit{if } x \neq y.\end{aligned}$$

Whenever needed, we will write $[x{:=}N]$ as syntactic sugar for $(\iota, x{:=}N)$.
Also, we often consider the *restriction* of a substitution σ to the free variables of a given term M, which is just written (σ, M). Then, we can extend fresh and free variables to restrictions as well:

Definition 2.4 *A variable x is fresh in (σ, M) and written $x\#(\sigma, M)$ iff for all $y * M$, then $x\#\sigma y$. On the contrary, x is said to be free in (σ, M) and written $x * (\sigma, M)$ iff there exists some $y * M$ such that $x * \sigma y$.*

Definition 2.5 (Effect of Substitution on Terms) *The effect of a substitution σ on a term M is defined by recursion on M:*

$$\begin{aligned}x \cdot \sigma &= \sigma x\\(MN)\cdot\sigma &= (M\cdot\sigma)(N\cdot\sigma)\\(\lambda x M)\cdot\sigma &= \lambda z(M\cdot(\sigma, x{:=}z))\quad \textit{with } z = \chi(\sigma, \lambda x M)\end{aligned}\tag{1}$$

In equation (1), notice that the bound variable x is always replaced with a new one. The new variable z is obtained by means of a choice function χ that computes the first variable fresh in $(\sigma, \lambda x M)$, so that it does not capture any of the names introduced into its scope by effect of the substitution (see [3] for more details). Besides implementing capture-avoidance, the use of multiple substitutions and uniform renaming of bound variables allows us to employ simple structural induction, avoiding case analyses when reasoning with substitutions. We will use $M\sigma$ as syntactic sugar for $M \cdot \sigma$ and, in concrete syntax, substitution will bind tighter than application. The following properties about substitutions are proven in [3]:

Lemma 2.6

1. $\chi(\sigma, M)\#(\sigma, M)$
2. $\chi(\iota, M)\#M$
3. $M(\sigma, x{:=}N\sigma) = M[x{:=}N]\sigma$

Definition 2.7 (Equality on Restrictions) $(\sigma, M) = (\sigma', M')$ *iff M and M' share the same free variables and $(\forall x * M)\ \sigma x = \sigma' x$.*

Then, $\sigma = \sigma' \downarrow M$ will be sugar for $(\sigma, M) = (\sigma', M)$.

2.3 Alpha Conversion

Definition 2.8 (Alpha-conversion)

$$\sim\texttt{v}\!:\ \frac{}{x \sim_\alpha x}\qquad \sim\texttt{app}\!:\ \frac{M \sim_\alpha M' \qquad N \sim_\alpha N'}{MN \sim_\alpha M'N'}\qquad \sim\lambda\!:\ \frac{M[x{:=}z] \sim_\alpha N[y{:=}z]}{\lambda x M \sim_\alpha \lambda y N}\ (*)$$

(*) with $z\#\lambda x M, \lambda y N$.

Alpha conversion is extended to restrictions by the following definition:

Definition 2.9 $\sigma \sim_\alpha \sigma' \downarrow M = (\forall x * M)\ \sigma x \sim_\alpha \sigma' x$

Then, the following lemmas –also proven in [3]– hold:

Lemma 2.10

1. $\iota \sim_\alpha \iota \downarrow M$
2. $\sigma \sim_\alpha \sigma' \downarrow M \wedge N \sim_\alpha P \Rightarrow (\sigma, x{:=}N) \sim_\alpha (\sigma', x{:=}P) \downarrow M$

Next we show some results about $\sim_\alpha$ proven in [3] and needed in our development:

Lemma 2.11

1. $\sim_\alpha$ *is an equivalence relation.*
2. $y\#(\sigma, \lambda x M) \Rightarrow M(\sigma, x{:=}y)[y{:=}N] \sim_\alpha M(\sigma, x{:=}N)$
3. $y\#\lambda x M \Rightarrow \lambda x M \sim_\alpha \lambda y(M[x{:=}y])$
4. $M \sim_\alpha N \Rightarrow \lambda x M \sim_\alpha \lambda x N$

5. $y\#(\sigma, \lambda xM) \Rightarrow (\lambda xM)\sigma \sim_\alpha \lambda y(M(\sigma, x{:=}y))$
6. $x\#M \wedge M \sim_\alpha N \Rightarrow x\#N$
7. $M \sim_\alpha M' \wedge \sigma \sim_\alpha \sigma' \downarrow M \Rightarrow M\sigma \sim_\alpha M'\sigma'$
8. $M \sim_\alpha N \Rightarrow M\sigma = N\sigma$
9. $M \sim_\alpha M\iota$

From these, some useful corollaries are immediate:

Corollary 2.12

1. $z\#(\sigma, \lambda yM) \Rightarrow M(\sigma, y{:=}z)[z{:=}N\sigma] \sim_\alpha M[y{:=}N]\sigma$
2. $(\lambda xM)N \sim_\alpha (\lambda yM')N' \Rightarrow M[x{:=}N] \sim_\alpha M'[y{:=}N']$
3. $\lambda xM \sim_\alpha \lambda yN \Rightarrow M[x{:=}y] \sim_\alpha N$

2.4 Beta Reduction

(One step) β-reduction is denoted $\rightarrow_\beta$, and it is defined by the following rules:

Definition 2.13 (Beta Reduction)

$$\beta\colon \frac{}{(\lambda xM)N \rightarrow_\beta M[x{:=}N]} \qquad \texttt{appL}\colon \frac{M \rightarrow_\beta M'}{MN \rightarrow_\beta M'N} \qquad \texttt{appR}\colon \frac{N \rightarrow_\beta N'}{MN \rightarrow_\beta MN'}$$

$$\lambda\colon \frac{M \rightarrow_\beta M'}{\lambda xM \rightarrow_\beta \lambda xM'}$$

Multi-step reduction is written $\twoheadrightarrow$ and defined below:

Definition 2.14 (Multi-step Reduction)

$$\frac{M \sim_\alpha N}{M \twoheadrightarrow N} \qquad \frac{M \rightarrow_\beta N}{M \twoheadrightarrow N} \qquad \frac{M \twoheadrightarrow N \qquad N \twoheadrightarrow P}{M \twoheadrightarrow P}$$

We extend $\twoheadrightarrow$ to substitutions as follows:

Definition 2.15 (Reductions on Substitutions) $\sigma \twoheadrightarrow \sigma' \ = (\forall x)\ \sigma x \twoheadrightarrow \sigma' x$.

The following results from [3] involving $\rightarrow_\beta$ will be needed in our development:

Lemma 2.16

1. $x\#M \wedge M \rightarrow_\beta N \Rightarrow x\#N$
2. $x\#(\sigma, M) \wedge M \rightarrow_\beta N \Rightarrow x\#(\sigma, N)$
3. $\sigma \twoheadrightarrow \sigma' \Rightarrow M\sigma \twoheadrightarrow M\sigma'$
4. $M \twoheadrightarrow M' \Rightarrow MN \twoheadrightarrow M'N$

From now on, except when explicitly stated, all results presented constitute new developments.

Lemma 2.17 (Compatibility of substitution with $\rightarrow_\beta$, up to $\sim_\alpha$)

$$M \rightarrow_\beta N \Rightarrow (\exists P)\ M\sigma \rightarrow_\beta P \sim_\alpha N\sigma$$

Proof. By induction on the derivation of $M \rightarrow_\beta N$.

- Case β: $(\lambda xM)N \rightarrow_\beta M[x{:=}N]$. Let $P = M(\sigma,\ x{:=}z)(\iota,\ z{:=}N\sigma)$ with $z = \chi(\sigma, \lambda xM)$. Then, we need to show: (i) $((\lambda xM)N)\sigma \rightarrow_\beta P$ and (ii) $P \sim_\alpha M[x{:=}N]\sigma$. (i) By definition of the effect of substitution (Def. 2.5) we have $((\lambda xM)N)\sigma = (\lambda z(M(\sigma,\ x{:=}z))N\sigma$ and by β rule, $(\lambda z(M(\sigma,\ x{:=}z))N\sigma \rightarrow_\beta M(\sigma,\ x{:=}z)(\iota,\ z{:=}N\sigma)$. (ii) By Corollary 2.12.1.
- Case `appL`: $MN \rightarrow_\beta M'N$ follows from $M \rightarrow_\beta M'$. By ind. hyp. we know there exists some Q s.t. $M\sigma \rightarrow_\beta Q \sim_\alpha M'\sigma$. Using `appL` rule we get $(MN)\sigma \rightarrow_\beta QN\sigma$ and by $\sim$`app` it follows $QN\sigma \sim_\alpha (M'N)\sigma$.
- Case `appR` : $MN \rightarrow_\beta MN'$ follows from $N \rightarrow_\beta N'$. Analogous to the `appL` case.
- Case λ: $\lambda xM \rightarrow_\beta \lambda xN$ follows from $M \rightarrow_\beta N$. We have to show: (i) $(\lambda xM)\sigma \rightarrow_\beta P$ and (ii) $P \sim_\alpha (\lambda xN)\sigma$, for some P. First, by ind. hyp. we know that there exists some P' s.t. $M(\sigma,\ x{:=}z) \rightarrow_\beta P' \sim_\alpha N(\sigma,\ x{:=}z)$. Let $P = \lambda zP'$. On the one side, (i) follows immediately by λ rule and Definition 2.5. (ii) On the other side,

by Lemma 2.11.4 we have $\lambda z P' \sim_\alpha \lambda z N(\sigma,\ x{:=}z)$. In addition, by Lemma 2.6.1 we get $z \# (\sigma, \lambda x M)$, thus by Lemma 2.16.2 we obtain $z\#(\sigma, \lambda x N)$. Finally, by Lemma 2.11.5 we have $\lambda z P' \sim_\alpha (\lambda x N)\sigma$, as desired.□

Lemma 2.18 (Commutativity of $\sim_\alpha$ and $\to_\beta$)

$$M \sim_\alpha N \to_\beta P \Rightarrow (\exists Q)\ M \to_\beta Q \sim_\alpha P$$

Proof. By induction on M. We show only the proofs of the interesting cases.

- Case $\lambda x M \sim_\alpha \lambda y N \to_\beta \lambda y P$. On the one hand, $\lambda y N \to_\beta \lambda y P$ must follow from $N \to_\beta P$ (Def. 2.13) and by Lemma 2.17 there is a term Q s.t. $N[y{:=}x] \to_\beta Q \sim_\alpha P[y{:=}x]$. In addition, by Corollary 2.12.3 we get $M \sim_\alpha N[y{:=}x]$. Then, by ind. hyp. using $M \sim_\alpha N[y{:=}x] \to_\beta Q$ we obtain $M \to_\beta Q' \sim_\alpha Q$, for some Q', and by Definition 2.13 we get $\lambda x M \to_\beta \lambda x Q'$. On the other hand, by hypothesis, using Lemmas 2.11.6 and 2.16.1 we get $x \# \lambda y P$, and by Lemma 2.11.3, $\lambda y P \sim_\alpha \lambda x P[y{:=}x]$. Finally, $\lambda x Q' \sim_\alpha \lambda y P$ holds by Lemma 2.11.4 and transitivity of $\sim_\alpha$.
- Case $(\lambda x M)M' \sim_\alpha (\lambda y N)N' \to_\beta N[y{:=}N']$. Let $Q = M[x{:=}M']$. By Def. 2.13 we get $(\lambda x M)M' \to_\beta Q$ and using Lemma 2.12.2 we have $Q \sim_\alpha N[y{:=}N']$. □

Lemma 2.19 (Commutativity of $\sim_\alpha$ and $\twoheadrightarrow$)

$$M \sim_\alpha N \twoheadrightarrow P \Rightarrow (\exists Q)\ M \twoheadrightarrow Q \sim_\alpha P$$

Proof. Follows easily by cases on $N \twoheadrightarrow P$, using Lemma 2.18. □

2.5 Unary Substitutions

It turns out convenient to introduce substitutions that arise in the process of reducing a β-redex. They will consist of a pair associating a variable to a term accompanied by several variable renamings.

Definition 2.20 *A substitution σ is unary of variable x and term M –written as* $\mathsf{Unary}\,\sigma\,x\,M$*– iff $\sigma x = M$ and, for all $y \neq x$, σy is a variable.*

Lemma 2.21 (Unary Substitution Lemmas)

1. $\mathsf{Unary}\,[x{:=}M]\,x\,M$
2. $\mathsf{Unary}\,\sigma\,x\,M \Rightarrow \mathsf{Unary}\,(\sigma, x{:=}N)\,x\,N$
3. $y \neq x \wedge \mathsf{Unary}\,\sigma\,x\,M \Rightarrow \mathsf{Unary}\,(\sigma, y{:=}z)\,x\,M$
4. $\mathsf{Unary}\,\sigma\,x\,y \Rightarrow \mathsf{Unary}\,(\sigma, z{:=}M)\,z\,M$

Proof. Trivially by definition. □

3 Inductive Definitions of Strongly Normalizing Terms

Our goal is to prove that all terms typable in the system of assignment of simple types are strongly normalizing. In order to do that, we will proceed as follows:
In this section we present two definitions of the strongly normalizing terms. The first one (predicate sn) is naturally introduced as the accessible part of the one-step beta reduction. The second one (predicate SN) is an alternative syntactic characterization introduced in [13], originally given in vector notation and adapted in [1] in a somewhat different context (i.e. using typed reductions).
We shall presently give the two definitions together with some of their properties, in order to eventually prove the soundness of the second definition with respect to the original one, i.e. that for all terms M, $\mathsf{SN}\,M$ implies $\mathsf{sn}\,M$, in the next section.
Later, in Section 5, we will complete the Strong Normalization proof by showing that all terms typable in the system of simple types assignment satisfy the second definition (SN).

3.1 Strongly Normalizing terms

A term is strongly normalizing if and only if it is in the accessible part of $\to_\beta$:

Definition 3.1 (Accessibility definition of strongly normalizing terms)

$$\frac{(\forall N)\ (M \to_\beta N \Rightarrow \mathsf{sn}\,N)}{\mathsf{sn}\,M}$$

Lemma 3.2 (Closure of sn under $\sim_\alpha$) $\mathsf{sn}\,M \wedge M \sim_\alpha N \Rightarrow \mathsf{sn}\,N$.

Proof. By induction on $\mathsf{sn}\, M$. Take N such that $M \sim_\alpha N$ and P such that $N \to_\beta P$. By Lemma 2.18 we have Q s.t. $M \to_\beta Q \sim_\alpha P$. Hence $\mathsf{sn}\, Q$ and by ind. hyp., since $Q \sim_\alpha P$, $\mathsf{sn}\, P$ as desired. □

Lemma 3.3 (Properties of sn)

1. $\mathsf{sn}\, x$
2. $\mathsf{sn}\, M \ \wedge\ M \twoheadrightarrow N \Rightarrow \mathsf{sn}\, N$
3. $\mathsf{sn}\, (MN) \Rightarrow \mathsf{sn}\, M \wedge \mathsf{sn}\, N$
4. $\mathsf{sn}\, M \Rightarrow \mathsf{sn}\, (\lambda x M)$

Proof. (3.3.1) By vacuity, since there is no possible way of having $x \to_\beta M$ for any x. (3.3.2) By induction on $M \twoheadrightarrow N$. (3.3.3) By induction on $\mathsf{sn}\, (MN)$. (3.3.4) By induction on $\mathsf{sn}\, M$. □

3.2 Strongly Normalizing terms: a syntactic definition

Now we introduce the second definition of strongly normalizing terms. In order to do this, we must define three mutually inductive predicates: strongly normalizing terms (SN), strongly normalizing neutral terms with head x (SNe_x) and the relation of strong head reduction ($\to_{\mathsf{SN}}$).

Definition 3.4 (SN, SNe **and** $\to_{\mathsf{SN}}$)

$$\mathtt{v}:\ \frac{}{\mathsf{SNe}_x\, x} \qquad \mathtt{app}:\ \frac{\mathsf{SNe}_x\, M \qquad \mathsf{SN}\, N}{\mathsf{SNe}_x\, MN}$$

$$\mathtt{ne}:\ \frac{\mathsf{SNe}_x\, M}{\mathsf{SN}\, M} \qquad \lambda:\ \frac{\mathsf{SN}\, M}{\mathsf{SN}\, \lambda x M} \qquad \mathtt{exp}:\ \frac{M \to_{\mathsf{SN}} N \qquad \mathsf{SN}\, N}{\mathsf{SN}\, M}$$

$$\beta:\ \frac{\mathsf{SN}\, N}{(\lambda x M)N \to_{\mathsf{SN}} M[x{:=}N]} \qquad \mathtt{appL}:\ \frac{M \to_{\mathsf{SN}} M'}{MN \to_{\mathsf{SN}} M'N}$$

$$\alpha:\ \frac{M \to_{\mathsf{SN}} R \qquad R \sim_\alpha N}{M \to_{\mathsf{SN}} N}$$

4 Soundness of SN

In this section we will show that SN is sound with respect to sn, i.e, every term in the first predicate is also in the second one. In order to achieve this, we first need to introduce two notions, namely that of neutral term and that of strongly normalizing head reductions that do not step inside abstractions.

4.1 Neutral terms

Let us call *neutral* the iterated applications having a variable as head:

Definition 4.1 (Neutral terms)

$$\frac{}{\mathsf{ne}_x\, x} \qquad \frac{\mathsf{ne}_x\, M}{\mathsf{ne}_x\, MN}$$

We will omit the head and just write $\mathsf{ne}\, M$ whenever it results convenient.
Neutral terms enjoy the following properties:

Lemma 4.2 (Closure of ne under $\to_\beta$) $\mathsf{ne}\, M \ \wedge\ M \to_\beta N \Rightarrow \mathsf{ne}\, N$

Proof. By induction on $\mathsf{ne}\, M$. □

Lemma 4.3 (Closure of sn/ne under app.) $\mathsf{ne}\, M \wedge \mathsf{sn}\, M \wedge \mathsf{sn}\, N \Rightarrow \mathsf{sn}\, MN$.
Proof. By lexicographic induction on $\mathsf{sn}\, M$, $\mathsf{sn}\, N$. Assume $MN \to_\beta P$ and proceed by cases to prove $\mathsf{sn}\, P$:

- In case $P = M'N$ with $M \to_\beta M'$, using Lemma 4.2 we obtain $\mathsf{ne}\, M'$. By definition of $\mathsf{sn}\, M$ we get $\mathsf{sn}\, M'$ and then, by ind. hyp., $\mathsf{sn}\, P$.
- In case $P = MN'$ with $N \to_\beta N'$ proceed analogously using $\mathsf{sn}\, N'$. □

4.2 Strongly Normalizing head reductions

Definition 4.4 (Strongly Normalizing head reductions) $\to_{\mathsf{sn}}$ *is the reduction relation defined as:*

$$\beta:\ \frac{\mathsf{sn}\, N}{(\lambda x M)N \to_{\mathsf{sn}} M[x{:=}N]} \qquad \mathtt{appL}:\ \frac{M \to_{\mathsf{sn}} M'}{MN \to_{\mathsf{sn}} M'N} \qquad \alpha:\ \frac{M \to_{\mathsf{sn}} R \qquad R \sim_\alpha N}{M \to_{\mathsf{sn}} N}$$

Lemma 4.5 (Confluence) *Let $M \rightarrow_{\mathsf{sn}} N$ and $M \rightarrow_\beta P$. Then either $N \sim_\alpha P$ or there exists Q s.t. $P \rightarrow_{\mathsf{sn}} Q$ and $N \twoheadrightarrow Q$.*

In other words, let Δ_L be the outer-leftmost redex of M not inside of an abstraction: then N is obtained from M by contracting Δ_L. Let Δ be the redex contracted in $M \rightarrow_\beta P$. If $\Delta = \Delta_L$, then $N = P$. Otherwise, let Q be obtained by contracting Δ_L from P, i.e. $P \rightarrow_{\mathsf{sn}} Q$. If Δ was discarded in $M \rightarrow_{\mathsf{sn}} N$, then $N = Q$. If not, then necessarily $N \rightarrow_\beta Q$ is derived by contracting Δ –and possibly other redexes in it.

Proof. By induction on $M \rightarrow_{\mathsf{sn}} N$ and subordinate analysis of $M \rightarrow_\beta P$.

- Case β: Suppose $\mathsf{sn}\, N$.
  - If $(\lambda x M)N \rightarrow_\beta P$ is obtained by rule β then we get $P = M[x{:=}N]$ immediately.
  - If $(\lambda x M)N \rightarrow_\beta P$ is obtained by rule `appL` then P is $(\lambda x M')N$ and $M \rightarrow_\beta M'$. Let $Q = M'[x{:=}N]$. By Lemma 2.17 and Definition 2.14 it follows that $M[x{:=}N] \twoheadrightarrow Q$ and then, using rule β of Definition 4.4, we get $P \rightarrow_{\mathsf{sn}} Q$.
  - If $(\lambda x M)N \rightarrow_\beta P$ is obtained by rule `appR` then P is $(\lambda x M)N'$ and $N \rightarrow_\beta N'$. Let $Q = M[x{:=}N']$. Now, on the one hand, by Lemma 2.17 we have $M[x{:=}N] \twoheadrightarrow Q$. And, on the other, since $\mathsf{sn}\, N'$ follows from $\mathsf{sn}\, N$ and Lemma 3.3.2, we can use again rule β of Definition 4.4 to get $P \rightarrow_{\mathsf{sn}} Q$.
- Case `appL`: Suppose $M \rightarrow_{\mathsf{sn}} M'$.
  - $MN \rightarrow_\beta P$ cannot be obtained using rule β, for in such case M would be an abstraction. But we have $M \rightarrow_{\mathsf{sn}} M'$ and $\rightarrow_{\mathsf{sn}}$ does not proceed on abstractions.
  - If $MN \rightarrow_\beta P$ follows from rule `appL` then P is $M''N$ and we have $M \rightarrow_\beta M''$. The ind. hyp. gives us two possibilities: either $M' \sim_\alpha M''$ or there exists some Q' such that $M' \twoheadrightarrow Q'$ and $M'' \rightarrow_{\mathsf{sn}} Q'$. In the first case, it follows immediately $M'N \sim_\alpha M''N$. Otherwise, let $Q = Q'N$, then by Definitions 4.4 and 2.14 we get $M'N \twoheadrightarrow Q$ and $M''N \rightarrow_{\mathsf{sn}} Q$.
  - If $MN \rightarrow_\beta P$ is obtained by rule `appR` then P is MN' and we have $N \rightarrow_\beta N'$. Choose then $Q = M'N'$ and use definitions of $\twoheadrightarrow$ and $\rightarrow_{\mathsf{sn}}$.
- Case α: Suppose $M \rightarrow_{\mathsf{sn}} R$ and $R \sim_\alpha N$. Suppose further $M \rightarrow_\beta P$. The ind. hyp. gives either $R \sim_\alpha P$ or there exists Q s.t. $R \twoheadrightarrow Q$ and $P \rightarrow_{\mathsf{sn}} Q$. In the first case, from $R \sim_\alpha N$ and $R \sim_\alpha P$ we obtain $N \sim_\alpha P$. In the other case, since $R \sim_\alpha N$, from $R \twoheadrightarrow Q$ it follows also $N \twoheadrightarrow Q$ by Lemma 2.19. □

4.3 The Proof of Soundness

In order to prove that SN implies sn, we also need corresponding proofs of soundness of SNe *and* $\rightarrow_{\mathsf{SN}}$ with respect to sn and $\rightarrow_{\mathsf{sn}}$. The difficulty lies in the case corresponding to the `exp` rule in the definition of SN, where $\mathsf{SN}\, M$ follows from $M \rightarrow_{\mathsf{SN}} N$ and $\mathsf{SN}\, N$. By ind. hyp. we have $M \rightarrow_{\mathsf{sn}} N$ and $\mathsf{sn}\, N$, and then we need a way of showing that the $\rightarrow_{\mathsf{sn}}$ relation is backward closed under sn. This is the goal of Lemma 4.8 below. In fact, what this lemma states is that $\rightarrow_{\mathsf{sn}}$ is also sound: if we have a $\rightarrow_{\mathsf{sn}}$ reduction from M to N and N is strongly normalizing, then M must also be. This reflects the idea behinds the $\rightarrow_{\mathsf{sn}}$ relation: it characterizes all computation strategies or possible reductions.
Coming back to Lemma 4.8, its proof goes by case analysis on the derivation of $M \rightarrow_{\mathsf{sn}} N$ plus three auxiliary results –one for each case (Lemmas 4.6, 3.3.3 and 4.7).

Next, we need to prove that $\mathsf{sn}\, N$ and $\mathsf{sn}\, M[x{:=}N]$ implies $\mathsf{sn}\, (\lambda x M)N$. However, –as in the previous lemma– since we are explicitly dealing with alpha-conversion, we need a stronger induction hypothesis, so we state the thesis is valid for any Q alpha-convertible with $M[x{:=}N]$:

Lemma 4.6 (Weak Head Expansion) *Let $\mathsf{sn}\, N$, $\mathsf{sn}\, Q$ and $Q \sim_\alpha M[x{:=}N]$. Then $\mathsf{sn}\, (\lambda x M)N$.*

Proof. By lexicographic induction on $\mathsf{sn}\, N$ and $\mathsf{sn}\, Q$. Assume $(\lambda x M)N \rightarrow_\beta P$ for some P and then proceed by cases proving $\mathsf{sn}\, P$:

- Case $P = M[x{:=}N]$. By Lemma 3.2 and hypothesis.
- Case $P = (\lambda x M')N$ with $\lambda x M \rightarrow_\beta \lambda x M'$, which in turn must follow from $M \rightarrow_\beta M'$. By Lemma 2.17, we have $M[x{:=}N] \rightarrow_\beta R \sim_\alpha M'[x{:=}N]$ for some R. Then, using Lemma 2.18 we obtain S such that $Q \rightarrow_\beta S \sim_\alpha R$. Hence, since $S \sim_\alpha M'[x{:=}N]$ by transitivity of $\sim_\alpha$, we obtain by ind. hyp. $\mathsf{sn}\, (\lambda x M')N$.
- Case $P = (\lambda x M)N'$, obtained from $N \rightarrow_\beta N'$. By Lemma 2.16.3, $M[x{:=}N] \twoheadrightarrow M[x{:=}N']$. Also, by Lemma 3.2 we get $\mathsf{sn}\, (M[x{:=}N])$. We apply Lemma 3.3.2 to obtain $\mathsf{sn}\, (M[x{:=}N'])$ and then, by ind. hyp. we conclude $\mathsf{sn}\, ((\lambda x M)N')$. □

Next, we introduce the following lemma which will be useful later on Lemma 4.8:

Lemma 4.7 *Let $M \rightarrow_{\mathsf{sn}} M'$ as well as M, N and $M'N$ be sn. Then $\mathsf{sn}\, MN$.*

Proof. By lexicographic induction on $\mathsf{sn}\, M$, $\mathsf{sn}\, N$. Assume some $MN \to_\beta P$ and proceed by cases showing $\mathsf{sn}\, P$:

- Case $P = M''N$ with $M \to_\beta M''$. By Lemma 4.5, either $M' \sim_\alpha M''$ or there exists some Q such that $M' \twoheadrightarrow Q$ and $M'' \to_{\mathsf{sn}} Q$. The first case follows from the hypothesis $\mathsf{sn}\,(M'N)$ and Lemma 3.2. In the second case, first notice that by Lemma 3.3.2 and by Definition 2.14 we get $\mathsf{sn}\,(QN)$. Then, by definition of $\mathsf{sn}\, M$, we have $\mathsf{sn}\, M''$ and can apply the ind. hyp. to conclude $\mathsf{sn}\, P$.
- Case $P = MN'$ with $N \to_\beta N'$. Use then the ind. hyp. and $\mathsf{sn}\,(M'N')$. □

Lemma 4.8 (Backward Closure of sn) *Let $M \to_{\mathsf{sn}} N$ and $\mathsf{sn}\, M$. Then $\mathsf{sn}\, M$.*

Proof. By induction on $M \to_{\mathsf{sn}} N$.

- Case β: Using Lemma 4.6.
- Case `appL`: Assume $M \to_{\mathsf{sn}} M'$ and $\mathsf{sn}\,(M'N)$. By Lemma 3.3.3 we have $\mathsf{sn}\, M'$ and $\mathsf{sn}\, N$. By ind. hyp. we have $\mathsf{sn}\, M$, and using Lemma 4.7, we get $\mathsf{sn}\,(MN)$.
- Case α: Use Lemma 3.2 and the ind. hyp. □

Now we only need two preparatory results:

Lemma 4.9 (Soundness of SNe with respect to ne) $\mathsf{SNe}_x\, M \Rightarrow \mathsf{ne}_x\, M$.
Proof. By a simple induction on $\mathsf{SNe}_x\, M$. □

Lemma 4.10 $M \to_{\mathsf{SN}} N \Rightarrow M \twoheadrightarrow N$

Proof. By induction on $M \to_{\mathsf{SN}} N$.

- Case α: $M \to_{\mathsf{SN}} N$ is obtained from $M \to_{\mathsf{SN}} P \sim_\alpha N$. By ind. hyp. we have $M \twoheadrightarrow N$. Also, by Definition 2.14 we get $P \twoheadrightarrow N$. Then, by transitivity of $\sim_\alpha$ we have $M \twoheadrightarrow N$.
- Case β. By Definition 2.14.
- Case `appL`: $MN \to_{\mathsf{SN}} M'N$ is obtained from $M \to_{\mathsf{SN}} M'$. By ind. hyp. we have $M \twoheadrightarrow N$ hence, by Lemma 2.16.4 we have $MN \twoheadrightarrow M'N$. □

Finally, we prove that the alternative definition of the strongly normalizing terms is sound with respect to the accessible definition:

Theorem 4.11 (Soundness of SN)

1. $\mathsf{SN}\, M \Rightarrow \mathsf{sn}\, M$
2. $\mathsf{SNe}_x\, M \Rightarrow \mathsf{sn}\, M$
3. $M \to_{\mathsf{SN}} N \Rightarrow M \to_{\mathsf{sn}} N$

Proof. By simultaneous induction, following Definition 3.4.

- Case `v`: By Lemma 3.3.1.
- Case `app`: Necessarily $M = PQ$ and $\mathsf{SNe}_x\,(PQ)$ is obtained from $\mathsf{SNe}_x\, P$ and $\mathsf{SN}\, Q$. Then, by ind. hyp. we have $\mathsf{sn}\, P$ and $\mathsf{sn}\, Q$, and by Lemma 4.9 we get $\mathsf{ne}\, P$. Finally, by Lemma 4.3 we obtain $\mathsf{sn}\, PQ$.
- Case `ne`: By ind. hyp.
- Case λ: By ind. hyp. followed by Lemma 3.3.4.
- Case `exp`: By ind. hyp. followed by Lemma 4.8.
- Case β: By ind. hyp. and Definition 4.4.
- Case `appL`: By ind. hyp. and Definition 4.4. □

5 Strong Normalization Theorem

In this section we present the proof of the Strong Normalization Theorem for typable terms in the simply typed Lambda Calculus.

5.1 Typing judgements

Definition 5.1 *Types are defined by the following grammar, starting from a category τ of ground types:*

$$\alpha, \beta ::= \tau \mid \alpha \to \beta.$$

We introduce an ordering between types in order to later perform complete induction on them:

Definition 5.2 *We write $\alpha \preceq \beta$ when the type α is a (structural) component of type β, and $\alpha \prec \beta$ when it is a proper component.*

Definition 5.3 *The system of assignment of simple types to lambda terms is the following:*

$$\vdash\mathtt{v}\colon \frac{x \in \Gamma}{\Gamma \vdash x : \Gamma x} \qquad \vdash\mathtt{a}\colon \frac{\Gamma \vdash M : \alpha \to \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash MN : \beta} \qquad \vdash\lambda\colon \frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x M : \alpha \to \beta}$$

where Γ stands for a context (implemented as a finite list) of variable declarations of the form $x_1 : \alpha_1, x_2 : \alpha_2, \ldots, x_n : \alpha_n$. In rule $\vdash \mathtt{v}$, $x \in \Gamma$ means that x is declared in Γ, and Γx is the (last) type declared for x in Γ. The following are proven in [3]:

Lemma 5.4 (Weakening) *Let $x\#M$ and $\Gamma \vdash M : \alpha$. Then $\Gamma, x : \beta \vdash M : \alpha$.*

Lemma 5.5 (Subject Reduction) $\Gamma \vdash M : \alpha \wedge M \twoheadrightarrow N \Rightarrow \Gamma \vdash N : \alpha$.

5.2 Typed Substitutions

Next, we introduce typed substitutions (w.r.t. the free variables of a term M):

Definition 5.6 *(σ, M) assigns to the variables in Γ terms of appropriate types under Δ —all of which is to be written $\sigma : \Gamma \to \Delta \downarrow M$— iff for all $x * M$ s.t. $x \in \Gamma$, we have $\Delta \vdash \sigma x : \Gamma x$.*

These are useful for stating that, if a term M is typable then $M\sigma$ is also typable, as the following lemma —proven in [3]— declares:

Lemma 5.7 *Let $\Gamma \vdash M : \alpha$ and $\sigma : \Gamma \to \Delta \downarrow M$. Then $\Delta \vdash M\sigma : \alpha$.*

Then, the following three lemmas –also proven in [3]– hold:

Lemma 5.8 *Let $x\#(\sigma, \lambda y M)$ and $\sigma : \Gamma \to \Delta \downarrow \lambda y M$.*
Then $(\sigma, y{:=}x) : (\Gamma, y{:}\alpha) \to (\Delta, x{:}\alpha) \downarrow M$.

Lemma 5.9 *Let $x\#M$ and $\sigma : \Gamma \to \Delta \downarrow M$.*
Then $(\sigma, x{:=}y) : (\Gamma, y{:}\alpha) \to (\Delta, x{:}\alpha) \downarrow M$.

Lemma 5.10 *Let $\Gamma \vdash M : \alpha$. Then $[x{:=}M] : (\Gamma, x{:}\alpha) \to \Gamma \downarrow M$.*

We end up this subsection with some useful general results.

Lemma 5.11 *Let $\mathtt{ne}_x\, M$ and $\Gamma \vdash x : \beta$. Then:*

1. $\Gamma \vdash M : \alpha \Rightarrow \alpha \preceq \beta$
2. $\Gamma \vdash M : \alpha \to \gamma \Rightarrow \alpha \prec \beta$

Proof. Immediate. □

Lemma 5.12 $u\#M \Rightarrow M\sigma = M(\sigma, u{:=}N)$

Proof. By definition of equality of restrictions (Def. 2.7). □

5.3 Strong Normalization

It is possible to prove by using a simple induction on the given type system that the typable terms are strongly normalizing . For this, we need to deal with the case of applications, that will be proved after the theorem (Lemma 5.14).

Theorem 5.13 (Strong Normalization) $\Gamma \vdash M : \alpha \Rightarrow \mathsf{SN}\, M$.

Proof. By induction on $\Gamma \vdash M : \alpha$.

- Case $\vdash\mathtt{v}$: By Definition 3.4.
- Case $\vdash\mathtt{a}$: $\Gamma \vdash MN : \alpha$ is obtained from $\Gamma \vdash M : \beta \to \alpha$ and $\Gamma \vdash N : \beta$. By ind. hyp. we get $\mathsf{SN}\, M$ and $\mathsf{SN}\, N$. Then, by Lemma 5.14 2 (ii) we have $\mathsf{SN}\, MN$.
- Case $\vdash\lambda$: By ind. hyp. and Definition 3.4. □

We may now turn our attention to the main lemma. As already said, its purpose is to establish that $\mathsf{SN}\, MN$ follows from $\mathsf{SN}\, M$ and $\mathsf{SN}\, N$ for typable M and N but, in order to obtain this, some additional results must be proven simultaneously.

Lemma 5.14 *Let* $\Gamma \vdash M : \alpha$, $\mathsf{SN}\, N$, $\sigma : \Gamma \rightarrow \Delta \downarrow M$, $\mathsf{Unary}\, \sigma\, x\, N$ *and* $\Gamma \vdash x : \beta$. *Then:*

1. $\mathsf{SNe}_y\, M \Rightarrow$ *(i)* $\mathsf{SN}\, M\sigma$, *(ii)* $y {\neq} x \Rightarrow \mathsf{SNe}_{\sigma y}\, M\sigma$ *and (iii)* $\alpha = \beta {\rightarrow} \gamma \Rightarrow \mathsf{SN}\, MN$.
2. $\mathsf{SN}\, M \Rightarrow$ *(i)* $\mathsf{SN}\, M\sigma$ *and (ii)* $\alpha = \beta {\rightarrow} \gamma \Rightarrow \mathsf{SN}\, MN$.
3. $M \rightarrow_{\mathsf{SN}} P \Rightarrow M\sigma \rightarrow_{\mathsf{SN}} P\sigma$.

Proof. The proof proceeds by complete induction on the structure of the type β, and subordinate induction on Definition 3.4 indexed by M. So, to begin with, take β and assume the statement for all of its proper components. We proceed to the subordinate induction:

- Case $\mathtt{v}$: Then $M = y$ and we have to show 1: (i) $\mathsf{SN}\, y\sigma$, (ii) $y {\neq} x \Rightarrow \mathsf{SNe}_{\sigma y}\, y\sigma$ and (iii) $\alpha = \beta {\rightarrow} \gamma \Rightarrow \mathsf{SN}\, yN$. As to (i), either $y = x$ or not. If true, then by Def. 2.20 we have $\sigma y = N$. If not, —also by same definition— $\sigma y = z$ is a variable. Either case: $\mathsf{SN}\, \sigma y$. As to (ii), if $y {\neq} x$ then in addition by Def. 3.4 we have $\mathsf{SNe}_z\, z$. As to (iii): by rule $\mathtt{app}$ of Def. 3.4 on $\mathsf{SNe}_y\, y$ and $\mathsf{SN}\, N$, followed by rule $\mathtt{ne}$.
- Case $\mathtt{app}$: Then $M = PQ$ and $\mathsf{SNe}_y\, PQ$ follows from $\mathsf{SNe}_y\, P$ and $\mathsf{SN}\, Q$, and $\Gamma \vdash PQ : \alpha$ follows from $\Gamma \vdash P : \delta \rightarrow \alpha$ and $\Gamma \vdash Q : \delta$. We need to show 1: (i) $\mathsf{SN}\, (PQ)\sigma$, (ii) $y {\neq} x \Rightarrow \mathsf{SNe}_{\sigma y}\, (PQ)\sigma$ and (iii) $\alpha = \beta {\rightarrow} \gamma \Rightarrow \mathsf{SN}\, (PQ)N$. First, by ind. hyp. we have $\mathsf{SN}\, P\sigma$ and $\mathsf{SN}\, Q\sigma$. Besides, by Lem. 5.7 we get $\Delta \vdash P\sigma : \delta \rightarrow \alpha$ and $\Delta \vdash Q\sigma : \delta$. Now, we have either $y = x$ or not. In the first case, by Lem. 4.9 we have $\mathtt{ne}_x\, P$, thus by Lem. 5.11.2 it follows $\delta \prec \beta$. Then, as to (i), we can apply the main ind. hyp. 2(ii) with $M = P\sigma$, $N = Q\sigma$ and $\beta = \delta$ and get $\mathsf{SN}\, P\sigma Q\sigma$ which equals to $\mathsf{SN}\, (PQ)\sigma$, as desired. On the contrary, if $y {\neq} x$ then by ind. hyp. (ii) we get $\mathsf{SNe}_{\sigma y}\, P\sigma$. Further, by definition of SN it follows $\mathsf{SNe}_{\sigma y}\, P\sigma Q\sigma$, thus $\mathsf{SN}\, (PQ)\sigma$. (ii) Just proven. (iii) Similar to case $\mathtt{v}$.
- Case $\mathtt{ne}$: Then $\mathsf{SN}\, M$ follows from $\mathsf{SNe}_y\, M$ and 2 is immediate by the main ind. hyp. 1 (i) and (iii).
- Case λ: Then $M = \lambda y P$ and $\mathsf{SN}\, \lambda y P$ follows from $\mathsf{SN}\, P$ and $\Gamma \vdash \lambda y P : \delta \rightarrow \epsilon$ from $\Gamma, y : \delta \vdash P : \epsilon$. We need to show 2: (i) $\mathsf{SN}\, (\lambda y P)\sigma$ and (ii) $\mathsf{SN}\, (\lambda y P)N$. As to (i), we have $(\lambda y P)\sigma = \lambda z P(\sigma, y{:=}z)$ with $z = \chi(\sigma, \lambda x P)$. We now proceed by cases: either $y = x$ or not. In the first case, before we can apply ind. hyp. with $\mathsf{SN}\, P$ and $(\sigma, x{:=}z)$ we need to show $\Gamma, x : \delta \vdash x : \beta$. However, this does not necessarily follow, since δ is rather arbitrary. Nevertheless, we can proceed us follows: let us fix some $u = \chi(\iota, P)$. First, by Lemma 2.6.1 we have $u \# (\iota, P)$ and thus by Lemma 2.6.2 get $u \# P$. Now, by Lemma 2.6.1 we have $z \# \lambda x P$ and then by Lemma 5.8 obtain $(\sigma, x{:=}z) : \Gamma, x : \delta \rightarrow \Gamma, z : \delta \downarrow P$, thus by Lemma 5.9 have $((\sigma, x{:=}\, z), u{:=}v) : \Gamma, x{:}\delta, u{:}\beta \rightarrow \Delta, z{:}\delta, v{:}\beta \downarrow P$ for any v. Besides, by Lemma 5.4 we have $\Gamma, x{:}\delta, u{:}\beta \vdash P : \epsilon$ and then by Def. 5.3 we get $\Gamma, x{:}\delta, u{:}\beta \vdash u : \beta$. In addition, by Lemma 2.21.2 we have $\mathsf{Unary}\, (\sigma, x{:=}z)\, x\, z$, thus by Lemma 2.21.4 we get $\mathsf{Unary}\, ((\sigma, x{:=}z), u{:=}v)\, u\, v$. Now, since $\mathsf{SN}\, v$, we can apply the ind. hyp. and get $\mathsf{SN}\, P((\sigma, x{:=}z), u{:=}v)$. Then, by Lemma 5.12 $P((\sigma, x{:=}z), u{:=}v) = P(\sigma, x{:=}z)$, and thus $\mathsf{SN}\, P(\sigma, x{:=}z)$. Finally, by the λ rule of Def. 3.4 we obtain $\mathsf{SN}\, (\lambda x P)\sigma$, as desired. Now if, on the other hand, $y \neq x$, then by Lemma 2.21.3 we have $\mathsf{Unary}\, (\sigma, y{:=}z)\, x\, N$ and thus we can apply the ind. hyp. to obtain $\mathsf{SN}\, P(\sigma, y{:=}z)$ and therefore derive $\mathsf{SN}\, (\lambda y P)\sigma$ in the same way. As to (ii), first notice that, since $\alpha = \beta \rightarrow \gamma$, it follows $\beta = \delta$, hence $\Gamma, y{:}\beta \vdash y : \beta$. Then, by Lemma 2.21.1 we have $\mathsf{Unary}\, [y{:=}N]\, y\, N$ and by Lem. 5.10 we have $[y{:=}N] : \Gamma, y{:}\beta \rightarrow \Gamma \downarrow P$, thus we can apply the ind. hyp. (i) and obtain $\mathsf{SN}\, P[y{:=}N]$. Lastly, by β rule of Def. 3.4 we obtain $(\lambda y P)N \rightarrow_{\mathsf{SN}} P[y{:=}N]$ and thus by $\mathtt{exp}$ rule we can derive $\mathsf{SN}\, (\lambda y P)N$.
- Case $\mathtt{exp}$: Then $\mathsf{SN}\, M$ follows from $M \rightarrow_{\mathsf{SN}} P$ and $\mathsf{SN}\, P$, and we need to show 2: (i) $\mathsf{SN}\, M\sigma$ and (ii) $\mathsf{SN}\, MN$. As to (i), by the main ind. hyp. 3 we have $M\sigma \rightarrow_{\mathsf{SN}} P\sigma$ and by ind. hyp. 2(i) $\mathsf{SN}\, P\sigma$, thus by rule $\mathtt{exp}$ we obtain $\mathsf{SN}\, M\sigma$. As to (ii), by ind. hyp. (ii) on $\mathsf{SN}\, P$ we have $\mathsf{SN}\, PN$, therefore by rule $\mathtt{appL}$ applied to $M \rightarrow_{\mathsf{SN}} P$ we get $MN \rightarrow_{\mathsf{SN}} PN$ and thus by $\mathtt{exp}$ we can derive $\mathsf{SN}\, MN$, as desired.
- Case β: Then $(\lambda x M)N \rightarrow_{\mathsf{SN}} M[x{:=}N]$ follows from $\mathsf{SN}\, N$ and we need to show 3: $((\lambda x M)N)\sigma \rightarrow_{\mathsf{SN}} M[x{:=}N]\sigma$. First, notice that by Def. 2.5 we have $((\lambda x M)N)\sigma = (\lambda z M(\sigma, x{:=}z))N\sigma$ with $z = \chi(\sigma, \lambda x M)$. In addition, by the main ind. hyp. 1(i) we have $\mathsf{SN}\, N\sigma$, thus by β rule of Def. 3.4 we have $(\lambda z M(\sigma, x{:=}z))N\sigma \rightarrow_{\mathsf{SN}} M(\sigma, x{:=}z)[z{:=}N\sigma]$. Furthermore, by Lem. 2.6.1 we get $z \# (\sigma, \lambda y M)$, hence by Cor. 2.12.1 we have $M(\sigma, x{:=}z)[z{:=}N\sigma] \sim_\alpha M[x{:=}N]\sigma$. Finally, we can use rule α of Def. 3.4 and obtain $((\lambda x M)N)\sigma \rightarrow_{\mathsf{SN}} M[x{:=}N]\sigma$.
- Case $\mathtt{appL}$: Then $MN \rightarrow_{\mathsf{SN}} M'N$ follows from $M \rightarrow_{\mathsf{SN}} M'$ and we must show 3: $(MN)\sigma \rightarrow_{\mathsf{SN}} (M'N)\sigma$. By ind. hyp. we have $M\sigma \rightarrow_{\mathsf{SN}} M'\sigma$, thus by $\mathtt{appL}$ rule we can apply $N\sigma$ on both sides and obtain $M\sigma N\sigma \rightarrow_{\mathsf{SN}} M'\sigma N\sigma$ which by Def. 2.5 equals to $(MN)\sigma \rightarrow_{\mathsf{SN}} (M'N)\sigma$.
- Case α: Then $M \rightarrow_{\mathsf{SN}} P$ follows form $M \rightarrow_{\mathsf{SN}} N$ and $N \sim_\alpha P$ and we need to show 3: $M\sigma \rightarrow_{\mathsf{SN}} P\sigma$. By ind. hyp. we have $M\sigma \rightarrow_{\mathsf{SN}} N\sigma$ and by Lem. 2.11.8 we get $N\sigma = P\sigma$, thus by Lemma. 2.11.1 we obtain

$N\sigma \sim_\alpha P\sigma$ and then by α rule of Def. 3.4 we can derive $M\sigma \to_{\mathsf{SN}} P\sigma$, as desired. □

6 Conclusions

The preceding work should be assessed in connection with the expectations stated in the Introduction. We assign relevance first of all to the complexity of the formalized version, both in Agda and in mathematical English.

Concerning the latter, we believe that this method makes it possible to come closer to a presentation in textbook style that does not hide details that need formalization. Of course, the main price paid has been the explicit handling of alpha-conversion; this has shown itself mainly in the formulation of the main definitions and results, which often must replace identity of terms by their alpha conversion, due to the renaming implicit in the effect of substitution on abstractions. As to additional, housekeeping lemmas, these seem to have been circumscribed to results of closure under alpha conversion. Also Lemma 5.12 is particularly noticeable. It could be replaced by a proof that Definition 3.4 is closed under alpha conversion.

Another aspect of the complexity has to do with the kind of proof methods required: when dealing with terms, we have been able to proceed by using only structural induction. In addition to this, we used of course induction on the various predicates and relations introduced, among which there is the somewhat complex mutual Definition 3.4 of Section 3, as well as the well-founded induction on types which was the main method in the proof we chose to formalize.

On the other hand, the size of the Agda code has by no means exploded: it is of about 600 lines, split almost in halves between the proof of soundness of the inductive characterization of the strongly normalizing terms of Section 3 and the theorem of strong normalization property of Section 5. The complexity of the development can also be assessed by the effort required: the first author completed the work as part of his Master's thesis in about 400 hours of work. This of course reveals a quite high cost for each Agda line of code.

Our pre-existing library providing the basics for the formalization of the calculus by the method chosen was only marginally updated. The required new results are shown in Section 2. The library was readily understood and made use of by the author of the formalization, despite his not having much prior experience in Agda (this was of only one tutorial course).

We believe we have made progress towards developing a corpus of formalized meta-theory of the Lambda Calculus somewhat along the lines of [9]. We also think that the present approach allows to aspire at achieving such a goal using the sort of explicit mathematical English we have used above, in turn supported by the Agda version.

This work should be compared to other formalizations of strong normalization, most notably those referred to in [1]. They employ variants of higher-order abstract syntax and of the de Bruijn notation, both of which of course dispense with alpha conversion. A full discussion is outside the scope of the present paper, but should be carried out. We should for that matter experiment with the full challenge posed in [1], using Kripke-style logical relations for proving strong normalization. The use of typed reduction will then force us to reformulate the calculus. We also have tried a formalization in Constructive Type Theory using Agda of principles of induction implementing the Barendregt variable convention –see [5,4], somewhat similarly to [12]. Again, we are still short of a full comparison in this respect.

References

[1] Abel, A., G. Allais, A. Hameer, B. Pientka, A. Momigliano, S. Schäfer and K. Stark, *POPLMark reloaded: Mechanizing proofs by logical relations*, Journal of Functional Programming **29** (2019).

[2] Copello, E., *Agda library for Formal metatheory of the lambda calculus using Stoughton's substitution*, Available at https://github.com/ernius/formalmetatheory-stoughton.

[3] Copello, E., N. Szasz and Álvaro Tasistro, *Formal metatheory of the lambda calculus using Stoughton's substitution*, Theoretical Computer Science **685** (2017), pp. 65 – 82.

[4] Copello, E., N. Szasz and Álvaro Tasistro, *Machine-checked Proof of the Church-Rosser Theorem for the Lambda Calculus Using the Barendregt Variable Convention in Constructive Type Theory*, Electronic Notes in Theoretical Computer Science **338** (2018), pp. 79 – 95, the 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017).
URL `http://www.sciencedirect.com/science/article/pii/S1571066118300720`

[5] Copello, E., Álvaro Tasistro, N. Szasz, A. Bove and M. Fernández, *Alpha-Structural Induction and Recursion for the Lambda Calculus in Constructive Type Theory*, Electronic Notes in Theoretical Computer Science **323** (2016), pp. 109 – 124.
URL `http://www.sciencedirect.com/science/article/pii/S1571066116300354`

[6] Copes, M., N. Szasz and A. Tasistro, *Formalization in Constructive Type Theory of the Standardization Theorem for the Lambda Calculus using Multiple Substitution*, in: F. Blanqui and G. Reis, editors, Proceedings of the 13th International Workshop on *Logical Frameworks and Meta-Languages: Theory and Practice,* Oxford, UK, 7th July 2018, Electronic Proceedings in Theoretical Computer Science **274** (2018), pp. 27–41.

[7] Curry, H. B. and R. Feys, "Combinatory Logic, Volume I," North-Holland, 1958, xvi+417 pp., second printing 1968.

[8] Joachimski, F. and R. Matthes, *Short Proofs of Normalization for the simply- typed λ-calculus, permutative conversions and Gödel's T*, Archive for Mathematical Logic **42** (2003), pp. 59–87.

[9] Nipkow, T. and S. Berghofer, *Fundamental Properties of Lambda-calculus* (2019), available from https://isabelle.in.tum.de/library/HOL/HOL-Proofs-Lambda/document.pdf.

[10] 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 (2007).

[11] Stoughton, A., *Substitution revisited*, Theoretical Computer Science **59** (1988), pp. 317–325.

[12] Urban, C. and C. Tasson, *Nominal Techniques in Isabelle/HOL*, in: R. Nieuwenhuis, editor, *Automated Deduction – CADE-20*, Lecture Notes in Computer Science **3632**, Springer Berlin Heidelberg, 2005 pp. 38–53.

[13] van Raamsdonk, F. and P. Severi, *On Normalisation*, Technical report, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1995).