# LFSA 2022

## The Seventeenth International Workshop on Logical and Semantic Frameworks, with Applications

September 23-24, 2022

https://lsfa2022.github.io/

Belo Horizonte, Brazil

## Program Committee

| | |
|---|---|
| Beniamino Accattoli | Inria & École Polytechnique, France |
| Sandra Alves | Universidade de Porto, Portugal |
| Carlos Areces | UNC, Argentina |
| Mauricio Ayala Rincón | Universidade de Brasília, Brazil |
| Haniel Barbosa | UFMG, Brazil |
| Mario R. Folhadela Benevides | UFF, Brazil |
| Alejandro Díaz-Caro | CONICET-Buenos Aires University and Quilmes University, Argentina |
| Amy Felty | University of Ottawa, Canada |
| Pascal Fontaine (co-chair) | University of Liège, Belgium |
| Edward Hermann Haeusler | PUC-Rio de Janeiro, Brazil |
| Delia Kesner | Université de Paris, France |
| Temur Kutsia | RISC/JKU Linz, Austria |
| Bruno Lopes | UFF, Brazil |
| Ian Mackie | Polytechnique, France, and University of Sussex, UK |
| Alexandre Madeira | Universidade de Aveiro, Portugal |
| Sérgio Marcelino | Instituto de Telecomunicações, Portugal |
| Mariano Moscato | National Institute of Aerospace, USA |
| Daniele Nantes (co-chair) | Universidade de Brasília, Brazil |
| Vivek Nigam | Huawei Munich Research Center, Germany |
| Carlos Olarte | Université Sorbonne Paris Nord, France |
| Mateus de Oliveira Oliveira | University of Bergen, Norway |
| Valeria de Paiva | Topos Institute, Berkeley, USA |
| Alberto Pardo | Universidad de la República, Uruguay |
| Elaine Pimentel | University College London, UK |
| Giselle Reis | CMU-Qatar, Qatar |
| Umberto Rivieccio | UFRN, Brazil |
| Camilo Rocha | Pontificia Universidad Javeriana - Cali, Colombia |
| Daniel Ventura | Universidade Federal de Goiás, Brazil |
| Petrucio Viana | UFF, Brazil |

## External reviewers

| | |
|---|---|
| Andreas Lööw | Imperial College London, UK |
| José Proença | Polytechnic Institute of Porto, Portugal |
| Deivid Vale | Radboud University Nijmegen, The Netherlands |

## Table of Contents

# Extending the Quantitative Pattern-Matching Paradigm

Sandra Alves
DCC-FCUP & CRACS
University of Porto, Porto, Portugal
sandra@fc.up.pt

Delia Kesner
Université de Paris, CNRS, IRIF, France
Institut Universitaire de France (IUF), France
kesner@irif.fr

Miguel Ramos
DCC-FCUP & LIACC
University of Porto, Porto, Portugal
jmiguelsramos@gmail.com

## 1 Introduction

Pattern-matching is a programming technique that provides an efficient way of decomposing and processing data, and is available in several modern programming languages and proof assistants. However, the study of programming languages semantics is usually based on the $\lambda$-calculus and some of the properties of this calculus do not translate directly to languages with matching primitives. For this reason, the study of the semantics of programming languages with pattern-matching features is usually based on pattern-calculi instead, which are formal calculi equipped with built-in patterns [2, 6, 11, 12, 13]. One such calculus is the pair pattern-calculus presented in [3], where the notion of $\lambda$-abstraction was generalized to functions of the form $\lambda p.t$, where $p$ is either a variable or a pair pattern specifying the expected structure of their arguments. As an example, the term $\lambda(x,y).x$ becomes a valid abstraction that expects a pair of the form $(t,u)$ as argument and yields the first projection $t$ of the pair. Even though this language is powerful enough to reason about some of the most interesting features of existing syntactical matching mechanisms, its lack of general data constructors (such as the ones for lists and trees) and absence of definition of functions by cases, leave this first approach far from being a realistic model for modern programming languages.

Quantitative type systems have been independently introduced in the framework of the $\lambda$-calculus by Gardner [9] and Kfoury [14], but the relevance of such systems regarding resource-aware consumption investigations remained unnoticed until it was highlighted in [4] and in de Carvalho's thesis in 2007 [7] (see also [8]), when its relation with linear logic [10] and quantitative relational models was explored. In [3], two resource-aware type systems for the pair pattern-calculus were presented. The first of those, called System $\mathscr{U}$, provides upper-bounds for the length of head normalization sequences plus the size of their corresponding normal forms by means of a non-idempotent intersection type system.

In this paper, we present an extension of the pair pattern-calculus in [3] called $\kappa$-calculus. The extension consists on generalizing the built-in pair patterns with constructor-based patterns and a more sophisticated pattern-matching mechanism. We also provide a resource-aware type system based on System $\mathscr{U}$, that provides upper-bounds for the length of weak head-termination sequences of terms of the $\kappa$-calculus, called System $\mathscr{G}$. Our main result is that typing in $\mathscr{G}$ characterizes weak head-termination for the terms in the $\kappa$-calculus. In fact, a term $t$ is typable if and only if $t$ is weak head-terminating, and the number of nodes of the typing derivation tree bounds the number of steps of the evaluation sequence of $t$.

## 2 The $\kappa$-calculus

In this work, we further generalize the notion of $\lambda$-abstraction to functions of the form $\lambda k.m$, where $k$ is a **multi-pattern** and $m$ is a **multi-term**. Multi-patterns can be a single **variable** $[\![x]\!]$ or consist of a set of distinct **constructor patterns** $[\![\mathtt{c_1}\ \vec{p_1},\ldots,\mathtt{c_n}\ \vec{p_n}]\!]$ that specifies, by cases, that the expected structure of their arguments should be a constructor term matching one of the constructor patterns in it. Multi-terms specify the set of continuations for a multi-pattern: each pattern $c_i\ \vec{p_i}$ in the multi-pattern is assigned a term $t_i$, for $1 \leq i \leq n$. As an example, the term $\lambda[\![\mathtt{p}(x,y),\mathtt{t}(x,y,z)]\!].[\![y,y]\!]$ expects either an argument of the form $\mathtt{p}(t,u)$ or of the form $\mathtt{t}(t,u,w)$, and yields the second projection $u$ of either constructor. The sets of **terms** and **list contexts** of the $\kappa$-calculus are given by the following grammars:

$$\begin{array}{rrcl}
\text{(Multi-Patterns)} & k & ::= & [\![x]\!] \mid [\![\mathtt{c_1}\ \vec{p_1},\ldots,\mathtt{c_n}\ \vec{p_n}]\!] \text{ where } n \geq 1 \\
\text{(Patterns)} & p & ::= & x \mid \mathtt{c}\ \vec{p} \\
\text{(Multi-Terms)} & m & ::= & [\![t_1,\ldots,t_n]\!] \text{ where } n \geq 1 \\
\text{(Terms)} & t,u & ::= & x \mid \mathtt{c}\ \vec{t} \mid \lambda k.m \mid tu \mid t[p\backslash u] \\
\text{(List Contexts)} & \mathscr{L} & ::= & \Box \mid \mathscr{L}[p\backslash t]
\end{array}$$

where $x,y,z,\ldots$ range over a countable set of variables, every pattern $p$ **is assumed to be linear** (*i.e.*, every variable appears at most once in $p$), and top-level **constructors of multipatterns are pairwise distinct**, i.e., for a multi-pattern $[\![\mathtt{c_1}\ \vec{p_1},\ldots,\mathtt{c_n}\ \vec{p_n}]\!]$, then $\mathtt{c_i} \neq \mathtt{c_j}$ for $1 \leq i,j \leq n$ and $i \neq j$.

Whenever we write $\mathtt{c}\ \vec{t}$ (resp. $\mathtt{c}\ \vec{p}$), we assume that the length of $\vec{t}$ (resp. $\vec{p}$) matches the arity of $c$. Finally, we are going to assume that every function $\lambda k.m$ consists of a multi-pattern $k$ and a multi-term $m$ with the same lengths. As usual, terms are considered modulo $\alpha$-conversion. Given a list context $\mathscr{L}$ and a term $t$, $\mathscr{L}\langle t\rangle$ denotes the term obtained by replacing the unique occurrence of $\Box$ in $\mathscr{L}$ by $t$, possibly allowing the capture of free variables of $t$.

The operational semantics of the $\kappa$-calculus is an extension of the well-known notion of weak head reduction for the $\lambda$-calculus [1] and is given by relation $\to_{\mathtt{wh}}$ in Figure 1. Given the one-step reduction relation $\to_{\mathtt{wh}}$, we use $\to^k_{\mathtt{wh}}$ $(k \geq 0)$ to denote the reflexive-transitive closure of $\to_{\mathtt{wh}}$, in fact, more specifically, the composition of $k$ $\mathtt{wh}$-steps.

$$\frac{\mathtt{dlc}(\mathscr{L}) \cap \mathtt{fv}(u) = \emptyset}{\mathscr{L}\langle\lambda[\![x]\!].[\![t]\!]\rangle u \to_{\mathtt{wh}} \mathscr{L}\langle t[x\backslash u]\rangle}\ (\beta_{\mathtt{abs}})$$

$$\frac{\mathtt{dlc}(\mathscr{L}) \cap \mathtt{fv}(c_i\ \vec{u}) = \mathtt{dlc}(\mathscr{L}') \cap \mathtt{fv}(t_i) = \emptyset}{(\mathscr{L}\langle\lambda[\![\mathtt{c_1}\ \vec{p_1},\ldots,\mathtt{c_n}\ \vec{p_n}]\!].[\![t_1,\ldots,t_n]\!]\rangle)(\mathscr{L}'\langle\mathtt{c_i}\ \vec{u}\rangle) \to_{\mathtt{wh}} \mathscr{L}'(\mathscr{L}\langle t_i\ [p_i\vec{\backslash}u]\rangle) \quad (1 \leq i \leq n)}\ (\beta_{\mathtt{case}})$$

$$\frac{}{t[x\backslash u] \to_{\mathtt{wh}} t\{x\backslash u\}}\ (\mathtt{es}) \qquad \frac{\mathtt{dlc}(\mathscr{L}) \cap \mathtt{fv}(t) = \emptyset}{t[\mathtt{c}\ \vec{p}\backslash\mathscr{L}\langle\mathtt{c}\ \vec{u}\rangle] \to_{\mathtt{wh}} \mathscr{L}\langle t\ [p\vec{\backslash}u]\rangle}\ (\mathtt{match})$$

$$\frac{t \to_{\mathtt{wh}} t' \qquad \neg\mathtt{isfun}(t)}{tu \to_{\mathtt{wh}} t'u} \qquad \frac{t \to_{\mathtt{wh}} t'}{t[p\backslash u] \to_{\mathtt{wh}} t'[p\backslash u]} \qquad \frac{t \not\to_{\mathtt{wh}} \qquad u \to_{\mathtt{wh}} u' \qquad p \neq x}{t[p\backslash u] \to_{\mathtt{wh}} t[p\backslash u']}$$

Figure 1: Weak Head Reduction Strategy

We use $t\{x\backslash u\}$ to denote the meta-level substitution operation that replaces all the free occurrences of $x$ in $t$ by the term $u$. We use the predicate $\mathtt{isfun}(t)$ when $t$ is of the form $\mathscr{L}\langle\lambda k.m\rangle$. When matching a term of the form $\mathtt{c}\ \vec{t}$ (such that $\vec{t}=(t_1,\ldots,t_n)$) with a pattern of the form $c\ \vec{p}$ (such that $\vec{p}=(p_1,\ldots,p_n)$), we are going to write $[\vec{p}\backslash t]$ for $[p_1\backslash t_1]\cdots[p_n\backslash t_n]$. We write $\mathtt{var}(p)$ (resp. $\mathtt{var}(\llbracket \mathtt{c_1}\ \vec{p_1},\ldots,\mathtt{c_n}\ \vec{p_n}\rrbracket)$) to denote the **set of variables** in the pattern $p$ (resp. multi-pattern $\llbracket \mathtt{c_1}\ \vec{p_1},\ldots,\mathtt{c_n}\ \vec{p_n}\rrbracket$). The **domain of a list context** is defined as $\mathtt{dlc}(\Box)=\emptyset$ and $\mathtt{dlc}(\mathscr{L}[p\backslash u])=\mathtt{dlc}(\mathscr{L})\cup\mathtt{var}(p)$.

Rule ($\beta_{\mathtt{abs}}$) fires the computation of terms by transforming the application of an abstraction with a variable multi-pattern to a term, into a closure of the single continuation. Rule ($\beta_{\mathtt{case}}$) does implicit case analysis by pattern-matching and fires the computation of term by transforming the application of an abstraction with a non-variable multi-pattern to a constructor, into a closure of the continuation of that matched pattern. Decomposition of patterns and terms is performed by means of (`match`), when a constructor pattern is matched against a constructor term. Substitution if performed by rule (`es`), *i.e.*, an explicit (simple) matching of the form $[x\backslash u]$ is executed. This form of syntactic pattern matching is very simple and does not consider any kind of failure rules, but it is expressive enough to specify the well-known mechanism of matching. Context closure is similar to the $\lambda$-calculus case, but not exactly the same. Indeed, reduction is performed on the left-hand side of applications and closures whenever possible. Otherwise, arguments of explicit matching operators must be reduced in order to unblock these operators, *i.e.*, in order to decompose $[p\backslash u]$ when $p$ is a constructor pattern but $u$ is still not a constructor. Notice, however, that when $u$ is already a constructor, reduction inside $u$ cannot take place at all, thus implementing a kind of lazy strategy for pattern matching.

Some **ill-formed** terms are neither redexes nor a desired result for a computation: applying a constructor to a term $(\mathscr{L}\langle\mathtt{c}\ \vec{t}\rangle)\ u$; applying a function of the form $\mathscr{L}\langle\lambda\llbracket \mathtt{c_1}\ \vec{p_1},\ldots,\mathtt{c_n}\ \vec{p_n}\rrbracket.m\rangle$ to a constructor of the form $\mathscr{L}'\langle\mathtt{c}\ \vec{t}\rangle$, such that $\mathtt{c}\neq\mathtt{c_i}$ for $1\leq i\leq n$; applying a function of the form $\mathscr{L}\langle\lambda\llbracket \mathtt{c_1}\ \vec{p_1},\ldots,\mathtt{c_n}\ \vec{p_n}\rrbracket.m\rangle$ to another function; matching a pattern of the form $\mathtt{c}\ \vec{p}$ with a constructor of the form $\mathtt{c}'\ \vec{t}$, such that $\mathtt{c}\neq\mathtt{c}'$; matching a pattern of the form $\mathtt{c}\ \vec{p}$ with a function. We call these ill-formed terms **clashes** and say that a term is **clash-free** if it does not **weak head-reduce** to a term containing a clash. A rewriting system raising a warning (*i.e.*, a **failure**) when detecting a **head-clash** has been defined in [5]. This allowed the authors to focus on the set of **head-clash-free normal forms**. In this work, the set of **clash-free normal forms**, which serves the same purpose, is described by the following grammars:

$$\begin{array}{rrcl} \text{(Clash-Free Normal Forms)} & \mathscr{F} & ::= & \lambda k.m \mid \mathtt{c}\ \vec{t} \mid \mathscr{F}[\mathtt{c}\ \vec{p}\backslash\mathscr{N}] \mid \mathscr{N} \\ \text{(Neutral Clash-Free Normal Forms)} & \mathscr{N} & ::= & x \mid \mathscr{N}\ t \mid \mathscr{N}[\mathtt{c}\ \vec{p}\backslash\mathscr{N}] \end{array}$$

We say that $t$ is **weak head-terminating** if there exists a clash-free normal form $u\in\mathscr{F}$ and an integer $k\geq 0$ such that $t\to_{\mathtt{wh}}^{k} u$.

**Example 2.1.** *In the following example, we start by reducing the argument until we reach a constructor and only then do we match the argument and continue from there:*

$$\begin{array}{rrcl} & (\lambda\llbracket \mathtt{c}(x,y)\rrbracket.\llbracket x\rrbracket)((\lambda\llbracket x\rrbracket.\llbracket x\rrbracket)(\mathtt{c}(x',y'))) & \to_{wh} & (\lambda\llbracket \mathtt{c}(x,y)\rrbracket.\llbracket x\rrbracket)(x[x\backslash\mathtt{c}(x',y')]) \\ \to_{wh} & (\lambda\llbracket \mathtt{c}(x,y)\rrbracket.\llbracket x\rrbracket)(\mathtt{c}(x',y')) & \to_{wh} & x[x\backslash x'][y\backslash y'] \\ \to_{wh} & x[x\backslash x'] & \to_{wh} & x' \end{array}$$

*In the next example, we have a function that accepts products and triples, and returns the second projec-*

*tion for both. It is being applied to a triple:*

$$\begin{array}{rrcl} & (\lambda\llbracket \mathtt{p}(x,y), \mathtt{t}(x,y,z)\rrbracket.\llbracket y,y\rrbracket)(\mathtt{t}(x',y',z')) & \to_{wh} & y[x\backslash x'][y\backslash y'][z\backslash z'] \\ \to_{wh} & y[x\backslash x'][y\backslash y'] & \to_{wh} & y'[x\backslash x'] \\ \to_{wh} & y' & & \end{array}$$

*Lastly, note that we cannot construct an abstraction of the form* $\lambda\llbracket c\ \vec{p}, c\ \vec{q}\rrbracket.\llbracket t,u\rrbracket$*, since we require top-level constructors to be pairwise distinct. But, we can have pattern-matching failures that are not caught at the top-level:*

$$\begin{array}{rrcl} & (\lambda\llbracket \mathtt{c_1}(\mathtt{c_2}(x,y),z)\rrbracket.\llbracket y\rrbracket)(\mathtt{c_1}(\mathtt{c_3}(x',y'),z')) & \to_{wh} & y[\mathtt{c_2}(x,y)\backslash \mathtt{c_3}(x',y')][z\backslash z'] \\ \to_{wh} & y[\mathtt{c_2}(x,y)\backslash \mathtt{c_3}(x',y')] & \not\to_{wh} & \end{array}$$

*However, this failure is due to a clash, which means that* $(\lambda\llbracket \mathtt{c_1}(\mathtt{c_2}(x,y),z)\rrbracket.\llbracket y\rrbracket)(\mathtt{c_1}(\mathtt{c_3}(x',y'),z'))$ *is* ***not*** *weak head-terminating.*

## 3 Type System $\mathscr{G}$

In this work, we present a quantitative type system, called $\mathscr{G}$, that extends System $\mathscr{U}$ to constructor types and uses the size of arbitrary type derivations to reason about time (length of evaluation sequences) and space (size of normal forms). More precisely, we will rely on the fact that, when $t$ weak head-reduces to $t'$, the size of the type derivation of $t'$ will be smaller than that of $t$, thus the size of type derivations will provide us an **upper-bound** for the length of the normalization sequence for $t$ plus the size of the its normal forms. The set of types is described by the following grammars:

$$\begin{array}{rrcl} \text{(Constructor Types)} & \mathscr{C} & ::= & \mathtt{c}\ \vec{\mathscr{A}} \\ \text{(Types)} & \sigma & ::= & \bullet \mid \mathscr{C} \mid \mathscr{A} \to \sigma \\ \text{(Multiset Types)} & \mathscr{A} & ::= & [\sigma_i]_{i\in I} \end{array}$$

Note that, because we include the constructor name $c$ in each constructor type $c\ \vec{\mathscr{A}}$, each constructor term will be assigned a unique constructor type matching its own constructor name. Whenever we write $c\ \vec{\mathscr{A}}$, we are going to assume that the length of $\vec{\mathscr{A}}$ matches the arity of $c$.

The **typing system** of the language is described in Figure 2 and can be seen as an extension of System $\mathscr{U}$ [3]: on the one hand we consider generic constructors, and on the other hand we integrate an extended notion of abstraction that is able to define functions by cases. We use $\Phi \triangleright \Gamma \vdash t : \sigma$ (resp. $\Phi \triangleright \Gamma \vdash t : \mathscr{A}$) to denote term type derivations ending with the sequent $\Gamma \vdash t : \sigma$ (resp. $\Gamma \vdash t : \mathscr{A}$), and $\Pi \triangleright \Gamma \Vdash p : \mathscr{A}$ to denote pattern type derivations ending with the sequent $\Gamma \Vdash p : \mathscr{A}$. The **size of a derivation** $\Phi$, denoted by $\mathtt{sz}(\Phi)$, is the number of all the typing rules used in $\Phi$, except (`many`) and (`match`), which are not counted.

Most of the rule for terms are straightforward. Rule ($\mathtt{abs}_k$) is used to type functions in general. But, when ($\mathtt{abs}_k$) is used to type a case function, the type of the whole function is the type of any one of its cases, i.e., not all of its cases need to be typable in order to type the whole case function. Rule (`match`) is used to type the explicit matching operator $t[p\backslash u]$ and can be seen as a combination of rules (`app`) and ($\mathtt{abs}_k$). Rule ($\mathtt{pat}_v$) is used when the pattern is a variable $x$, and its multiset type is the type declared for $x$ in the typing context. Rule ($\mathtt{pat}_c$) is used when the pattern has a constructor type, which means that the pattern will be matched with a constructor. Rules (`constC`) and (`absC`) are used to type normal forms, when these are constructors and abstractions. Finally, note that when assigning types (multiset types) to terms, we only allow the introduction of multiset types on the right through the (`many`) rule.

**Example 3.1.** *In the following example, a typing derivation $\Phi$ for the first term in Example 2.1 is built. Let $\Phi_1$ be the following type derivation:*

$$\dfrac{\dfrac{}{x:[\sigma]\vdash x:\sigma}\,(ax) \qquad \dfrac{\dfrac{}{x:[\sigma]\vdash x:[\sigma]}\,(pat_v) \quad \dfrac{}{y:[\,]\vdash x:[\,]}\,(pat_v)}{x:[\sigma],y:[\,]\vdash \mathtt{c}(x,y):[\mathtt{c}([\sigma],[\,])]}\,(pat_c)}{\emptyset\vdash \lambda\llbracket \mathtt{c}(x,y)\rrbracket.\llbracket x\rrbracket : [\mathtt{c}([\sigma],[\,])]\to\sigma}\,(abs_k)$$

*Let $\Phi_2$ be the following type derivation:*

$$\dfrac{\dfrac{}{x:[\mathtt{c}([\sigma],[\,])]\vdash x:\mathtt{c}([\sigma],[\,])}\,(ax) \qquad \dfrac{}{x:[\mathtt{c}([\sigma],[\,])]\Vdash x:[\mathtt{c}([\sigma],[\,])]}\,(pat_v)}{\emptyset\vdash\lambda\llbracket x\rrbracket.\llbracket x\rrbracket : [\mathtt{c}([\sigma],[\,])]\to\mathtt{c}([\sigma],[\,])}\,(abs_k)$$

*And $\Phi_3$ be the following type derivation:*

$$\dfrac{\dfrac{\Phi_2 \qquad \dfrac{\dfrac{\dfrac{}{x':[\sigma]\vdash x':\sigma}\,(ax) \quad \dfrac{}{\vdash y':[\,]}\,(many)}{x':[\sigma]\vdash \mathtt{c}(x',y'):\mathtt{c}([\sigma],[\,])}\,(const)}{x':[\sigma]\vdash \mathtt{c}(x',y'):[\mathtt{c}([\sigma],[\,])]}\,(many)}{x':[\sigma]\vdash(\lambda\llbracket x\rrbracket.\llbracket x\rrbracket)(\mathtt{c}(x',y')):\mathtt{c}([\sigma],[\,])}\,(app)}{x':[\sigma]\vdash(\lambda\llbracket x\rrbracket.\llbracket x\rrbracket)(\mathtt{c}(x',y')):[\mathtt{c}([\sigma],[\,])]}\,(many)$$

*Then we can construct $\Phi$ as follows:*

$$\dfrac{\Phi_1\triangleright\emptyset\vdash\lambda\llbracket\mathtt{c}(x,y)\rrbracket.\llbracket x\rrbracket:[\mathtt{c}([\sigma],[\,])]\to\sigma \qquad \Phi_3\triangleright x':[\sigma]\vdash(\lambda\llbracket x\rrbracket.\llbracket x\rrbracket)(\mathtt{c}(x',y')):[\mathtt{c}([\sigma],[\,])]}{x':[\sigma]\vdash(\lambda\llbracket\mathtt{c}(x,y)\rrbracket.\llbracket x\rrbracket)((\lambda\llbracket x\rrbracket.\llbracket x\rrbracket)(\mathtt{c}(x',y'))):\sigma}\,(app)$$

*But, note that we cannot construct a type derivation for the last term in Example 3.1. Let $\Phi_1$ now be the following type derivation:*

$$\dfrac{\dfrac{}{y:[\sigma]\vdash y:\sigma}\,(ax) \qquad \dfrac{\dfrac{\dfrac{}{x:[\,]\Vdash x:[\,]}\,(pat_v) \quad \dfrac{}{y:[\sigma]\Vdash y:[\sigma]}\,(pat_v)}{y:[\sigma]\Vdash \mathtt{c}_2(x,y):[\mathtt{c}_2([\,],[\sigma])]}\,(pat_c) \quad \dfrac{}{z:[\,]\Vdash z:[\,]}\,(pat_v)}{y:[\sigma]\Vdash \mathtt{c}_1(\mathtt{c}_2(x,y),z):[\mathtt{c}_1([\mathtt{c}_2([\,],[\sigma])],[\,])]}\,(pat_c)}{\emptyset\vdash\lambda\llbracket\mathtt{c}_1(\mathtt{c}_2(x,y),z)\rrbracket.\llbracket y\rrbracket:[\mathtt{c}_1([\mathtt{c}_2([\,],[\sigma])],[\,])]\to\sigma}\,(abs_k)$$

*And $\Phi_2$ be the following type derivation:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash x':[\,]}\,(many) \quad \dfrac{\dfrac{}{y':[\sigma]\vdash y':\sigma}\,(ax)}{y':[\sigma]\vdash y':[\sigma]}\,(many)}{y':[\sigma]\vdash \mathtt{c}_3(x',y'):\mathtt{c}_3([\,],[\sigma])}\,(const) \quad \dfrac{}{\vdash z':[\,]}\,(many)}{y':[\sigma]\vdash \mathtt{c}_1(\mathtt{c}_3(x',y'),z'):\mathtt{c}_1(\mathtt{c}_3([\,],[\sigma]),[\,])}\,(const)}{y':[\sigma]\vdash \mathtt{c}_1(\mathtt{c}_3(x',y'),z'):[\mathtt{c}_1(\mathtt{c}_3([\,],[\sigma]),[\,])]}\,(many)$$

*Note that we cannot construct a type derivation for the aforementioned term, because the type of the argument will never match the type of the term expected by the function.*

We now state the main result of this work as the following theorem.

**Theorem 3.1** (Characterization of Weak Head Termination and upper-bounds)**.** *Let $t$ be a term in the pattern-calculus. Then $t$ is typable in system $\mathscr{G}$ iff $t$ is **weak head-terminating**. Moreover, if $\Phi\triangleright\Gamma\vdash t:\sigma$, then the **weak head strategy** terminates on $t$ in at most $sz(\Phi)$ steps.*

We will simply present a sketch of the proof, since we are omitting the necessary lemmas that are needed in order to present the whole proof.

$$\frac{}{x : \mathscr{A} \Vdash x : \mathscr{A}}\ (\mathtt{pat}_v)$$

$$\frac{(\Gamma_i \Vdash p_i : \mathscr{A}_i)_{1\leq i\leq n} \qquad \mathtt{c}(p_1,\ldots,p_n) \text{ is linear}}{\bigwedge_{1\leq i\leq n}\Gamma_i \Vdash \mathtt{c}(p_1,\ldots,p_n) : [\mathtt{c}(\mathscr{A}_1,\ldots,\mathscr{A}_n)]}\ (\mathtt{pat}_c)$$

---

$$\frac{}{x : [\sigma] \vdash x : \sigma}\ (\mathtt{ax}) \qquad \frac{}{\emptyset \vdash \mathtt{c}(t_1,\ldots,t_n) : \bullet}\ (\mathtt{constC})$$

$$\frac{(\Gamma_i \vdash t_i : \mathscr{A}_i)_{1\leq i\leq n}}{\bigwedge_{1\leq i\leq n}\Gamma_i \vdash \mathtt{c}(t_1,\ldots,t_n) : \mathtt{c}(\mathscr{A}_1,\ldots,\mathscr{A}_n)}\ (\mathtt{const})$$

$$\frac{}{\emptyset \vdash \lambda\llbracket p_1,\ldots,p_n\rrbracket.\llbracket t_1,\ldots,t_n\rrbracket : \bullet}\ (\mathtt{absC})$$

$$\frac{\Gamma_k \vdash t_k : \sigma_k \qquad \Gamma_k|_{p_k} \Vdash p_k : \mathscr{A}_k}{\Gamma_k \backslash\backslash \mathtt{var}(p_k) \vdash \lambda\llbracket p_1,\ldots,p_n\rrbracket.\llbracket t_1,\ldots,t_n\rrbracket : \mathscr{A}_k \to \sigma_k \quad (1\leq k\leq n)}\ (\mathtt{abs}_k)$$

$$\frac{\Gamma \vdash t : \mathscr{A} \to \sigma \qquad \Delta \vdash u : \mathscr{A}}{\Gamma \wedge \Delta \vdash tu : \sigma}\ (\mathtt{app}) \qquad \frac{(\Gamma_i \vdash t : \sigma_i)_{i\in I}}{\bigwedge_{i\in I}\Gamma_i \vdash t : [\sigma_i]_{i\in I}}\ (\mathtt{many})$$

$$\frac{\Gamma \vdash t : \sigma \qquad \Gamma|_p \Vdash p : \mathscr{A} \qquad \Delta \vdash u : \mathscr{A}}{(\Gamma \backslash\backslash \mathtt{var}(p)) \wedge \Delta \vdash t[p\backslash u] : \sigma}\ (\mathtt{match})$$

Figure 2: Typing System $\mathscr{G}$

*Proof.* The implication ($\Rightarrow$) holds by weighted subject reduction. That is, by the fact that if for two typable terms $t,u$, if $t \to_{\mathtt{wh}} u$, the size of the type derivation of $u$ is strictly smaller than the size of the type derivation of $u$. The implication ($\Leftarrow$) holds by the typability of clash-free normal forms and weighted subject expansion. That is, by the fact that if $u \in \mathscr{F}$, then $u$ is typable. And for another typable $t$ such that $t \to_{\mathtt{wh}} u$, then the size of the type derivation of $u$ is strictly smaller than the size of the type derivation of $t$. □

# 4 Future Work

In this paper, we have presented a pattern-calculus named $\kappa$-calculus that extends the $\lambda$-calculus with constructors and pattern-matching, and a resource-aware type system named $\mathscr{G}$ that provides upper-bounds for the length of weak head-termination sequences. In the future, we would like to explore the use of tight typings to provide exact bounds to the length of evaluation sequences and the size of normal forms. Additionally, we would like to extend this calculus to programs with recursive schemes that would allow us to not only define but also process recursive data structures such as lists and trees. Last, but not least, we expect this language to provide a good model for the study of AC-properties of pattern-matching for functional programming languages with built-in pattern-matching.

# References


[1] Samson Abramsky. The Lazy Lambda Calculus, page 65–116. Addison-Wesley Longman Publishing Co., Inc., USA, 1990.

[2] Sandra Alves, Besik Dundua, Mário Florido, and Temur Kutsia. Pattern-based calculi with finitary matching. Logic Journal of the IGPL, 26(2):203–243, December 2017.

[3] Sandra Alves, Delia Kesner, and Daniel Ventura. A Quantitative Understanding of Pattern Matching. In Marc Bezem and Assia Mahboubi, editors, 25th International Conference on Types for Proofs and Programs (TYPES 2019), volume 175 of Leibniz International Proceedings in Informatics (LIPIcs), pages 3:1–3:36, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[4] Gérard Boudol, Pierre-Louis Curien, and Carolina Lavatelli. A semantics for lambda calculi with resources. Mathematical Structures in Computer Science, 9(4):437–482, 1999.

[5] Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. Observability for pair pattern calculi. 2015.

[6] H Cirstea. The rewriting calculus - part i. Logic Journal of IGPL, 9(3):339–375, May 2001.

[7] Daniel de Carvalho. Sémantiques de la logique linéaire et temps de calcul. 2007.

[8] Daniel de Carvalho. Execution time of lambda-terms via denotational semantics and intersection types. CoRR, abs/0905.4251, 2009.

[9] Philippa Gardner. Discovering needed reductions using type theory. In TACS, 1994.

[10] Jean-Yves Girard. Linear logic. Theoretical Computer Science, 50(1):1–101, 1987.

[11] Barry Jay and Delia Kesner. First-class patterns. Journal of Functional Programming, 19(2):191–225, March 2009.

[12] Wolfram Kahl. Basic pattern matching calculi: a fresh view on matching failure. In Functional and Logic Programming, pages 276–290. Springer Berlin Heidelberg, 2004.

[13] Delia Kesner, Carlos Lombardi, and Alejandro Ríos. A standardisation proof for algebraic pattern calculi. Electronic Proceedings in Theoretical Computer Science, 49:58–72, February 2011.

[14] A. Kfoury. A linearization of the lambda-calculus and consequences. Journal of Logic and Computation, 10(3):411–436, June 2000.

# Towards a proof in Lean about the Horizontal Compression of Dag-Like Derivations in Minimal Purely Implicational Logic

Robinson Callou de Moura Brasil Filho
Informatics PUC-Rio
Rio de Janeiro, Brazil
rfilho@inf.puc-rio.br

Jefferson de Barros Santos
EBAPE/FGV
Rio de Janeiro, Brazil
jefferson.santos@fgv.br

Edward Hermann Haeusler
Informatics PUC-Rio
Rio de Janeiro, Brazil
hermann@inf.puc-rio.br

In this article, we argue that a dag-like proof can be obtained by compressing Natural Deduction proofs of tautologies in purely implicational minimal logic ($M_{\supset}$). We call these dag-like proofs as *DLDS*, for **D**ag-**L**ike **D**erivability **S**tructure. After compression, the size of a resulting dag-like proof $\Pi$ of an $M_{\supset}$ tautology $\alpha$ is $\mathcal{O}(h.m^4)$, where $h$ is the height of $\Pi$ and $m$ is the size of the set of all $M_{\supset}$ formulas occurring in $\Pi$. We call the compression algorithm as **HC**, for the horizontal (left-to-right) compression procedure we applied. This procedure follows a horizontal collapse from one valid *DLDS* to another, via a finite set of compression rules. As of the writing of this article, there are a total of 26 compression rules in **HC**, each with its own conditions. Because of this high number of rules and definitions, the manual and detailed proofs of some properties of **HC**, such as the fact that it preserves the soundness of the tree-like proofs or that it halts for every *DLDS* entry, is not easy to follow. The main purpose of this article is to present some properties of **HC** and the beginnings of a Lean-assisted proof showing that **HC** halts for every $M_{\supset}$ tautology, exiting a valid *DLDS* with no two equal nodes (vertexes) on the same level.

## 1 Introduction

Under the standard terminology of proof theory, a mathematical proof in Natural Deduction is a derivation without open assumptions. Each of its hypotheses must be discharged by applying a specific rule. In the case of purely implicational minimal logic ($M_{\supset}$), the only rules applicable are the $\supset$-Introduction (or $\supset$-Intro) and $\supset$-Elimination (or $\supset$-Elim) rules. These Natural Deduction rules of $M_{\supset}$ are shown below, according to [6]:

$$\dfrac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \supset B}\ \supset\text{-Introduction} \qquad\qquad \dfrac{A \qquad A \supset B}{B}\ \supset\text{-Elimination}$$

In [4], the authors provide an algorithm, named **HC**, for obtaining a compressed dag-like proof, meaning a proof represented by a **D**irected **A**cyclic **G**raph, of any purely implicational minimal tautology. This dag-like proof has more decoration elements and labels than regular proofs in purely implicational minimal logic ($M_{\supset}$) and, using these elements, a verification that the dag-like proof is valid can be done in

polynomial time [4]. The authors named this type of dag-like proof a **D**ag-**L**ike **D**erivability **S**tructure (*DLDS*), which is defined in [4] and also in accordance to the following definitions below, which we write down here in order to have a more self-contained document.

In any $M_{\supset}$ Natural Deduction derivation, any application of an $\supset$-introduction rule has some kind of mechanism to indicate which formula occurrences are discharged by the application of this $\supset$-introduction application. One way to formalize this indication is to add edges (discharging edges) linking the conclusion of the rule application to each discharged formula occurrence in the derivation tree that represents the Natural Deduction derivation[1]. This may be a convenient representation in many formulations of Natural Deduction. However, in order to not crowd our dag-like derivations with unnecessary edges, we drop out the discharging edges by assigning to each deduction edge the string of bits that represents the set of assumptions from which the formula that labels the target of this deduction edge depends on. This is formalized in the sequel.

➤ **Definition 1:** Let $\alpha$ be an implicational formula, $Sub(\alpha)$ the set of subformulas of $\alpha$, and $\mathscr{O}(\alpha) = \{\beta_0, \beta_1, \ldots, \beta_k\}$ a linear ordering on $Sub(\alpha)$. A bit-string on $\mathscr{O}(\alpha)$ is any string $b_0 b_1 \ldots b_k$, such that $b_i \in \{0,1\}$, for each $i = 0, 1, \ldots, k$.

There is a bijective correspondence between bit-strings on $\mathscr{O}(\alpha)$ and sets of subformulas of $\alpha$, given by $Set(b_0 b_1 \ldots b_k) = \{\beta_i / b_i = 1\}$. The bit-string on $\mathscr{O}(\alpha)$ will be used to drop out the discharging edges and make explicit the information on formula dependencies in a derivation. The set of bit-strings on $\mathscr{O}(\alpha)$ is denoted by $Bits(\alpha, \mathscr{O}(\alpha))$. The inverse function of $Set$ is well-defined, for a fix ordering on $Sub(\alpha)$ and is denoted by $Set^{-1}$. The set of all bit-strings on a set $S$, under ordering $\mathscr{O}_S$ is denoted by $\mathscr{B}(\mathscr{O}_S)$.

The following result shows that when considering a restricted form of $\supset$-Introduction rules, the set of theorems is not changed. This restricted form of an $\supset$-Introduction is used to provide a sound way to remove the discharging edges from the tree-like proofs.

➤ **Definition 2:** Consider a derivation $\Pi$ of $\beta$ having $\Delta$ as assumptions. Let $\alpha \in \Delta$ be a (open) formula assumption in $\Pi$. An application of an $\supset$-Introduction in $\Pi$ is greedy, if and only if, it produces $\alpha \supset \beta$ as conclusion and discharges in $\Pi$ every open occurrence of $\alpha$ from which its premise $\beta$ depends on.

An application of an $\supset$-Introduction in a tree-like derivation is greedy, if and only if, its corresponding application in a Natural Deduction derivation is also greedy. We reaffirm the terminology used in [6] that a proof is a derivation that has no open assumption, i.e., all hypotheses are discharged by some $\supset$-Introduction. The algorithm below modifies a given proof in $M_{\supset}$ into a greedy proof in $M_{\supset}$:

**Algorithm 1** Greedy Proof Conversion

**Require:** A proof $\Pi$ in $M_{\supset}$, where $n$ is the level of the highest branch in $\Pi$

```
j = n
while 0 < j do
    for each branch B of level j in Π do
        replace each intro-app downwards by a greedy intro-app
        possibly discharging more formula occurrences in B
    end for
    j = j − 1
end while
```

[1] It can be noted that assigning unique marks (numbers for example) to each formula occurrence in a derivation and attach to each $\supset$-Introduction application the set of marks associated to the set of its discharged formula occurrences is equivalent to add edges indicating these discharges.

In [4] the authors demonstrate the following lemma about the above procedure:

➤ **Lemma 1 (Greedy $\supset$-Introduction is Complete):** Let $\Pi$ be a proof of an $M_{\supset}$ formula $\alpha$. If $\Pi'$ is the result of the above procedure applied to $\Pi$, then $\Pi'$ is also a valid proof of $\alpha$ in $M_{\supset}$.

In [4] the authors also prove that any greedy proof of $\alpha$ is mapped to a rooted, leveled, and labeled dag-like proof of $\alpha$, where its root is labeled with $\alpha$. The tree-like dependency is also mapped into this initial dag-like proof. The article also argues that **HC**, when applied to any dag-like proof, preserves the logical information provided by the decorations used in the dag, resulting in the preservation of the soundness of the dag-like proof. However, this demonstration is lengthy and must go through multiple cases. A computer assisted proof seems to be the most appropriate way of proving the soundness of these rules. This article brings the beginnings of this proof of the soundness preservation of the compression algorithm, done with the Lean interactive theorem prover [3][2].

## 2 Horizontal Compression HC

### 2.1 Primary Definitions

In [4] (sections 3 and 5), in a series of hand-proven results, the authors show that any greedy proof of an $M_{\supset}$ formula $\alpha$ can be mapped to a *DLDS* having the root labeled by $\alpha$. In fact, this *DLDS* is the underlying tree of the Natural Deduction greedy proof, instanced with the decorations that a *DLDS* need. Since this article focuses on the proof of properties on the horizontal compression algorithm **HC** and the beginnings of the formalization of those properties in Lean, some of them will be omitted here. For our purposes, **HC** takes as input an arbitrary *DLDS*, defined as follows:

➤ **Definition 3 (Dag-Like Derivability Structures):** Let $\Gamma$ be a set of $M_{\supset}$ formulas, $\mathscr{O}_\Gamma$ an arbitrary linear ordering on $\Gamma$[2], and $\mathscr{O}^0_\Gamma = \mathscr{O}_\Gamma \cup \{0, \lambda\}$. A **D**ag-**L**ike **D**erivability **S**tructure, *DLDS* for short, is a tuple $\langle V, (E^i_D)_{i\in\mathscr{O}^0_\Gamma}, E_A, r, l, L, P\rangle$, where:

- ➥ $V$ is a non-empty set of nodes;
- ➥ For each $i \in \mathscr{O}^0_\Gamma$, $E^i_D \subseteq V \times V$ is the family of sets of edges of deduction;
- ➥ $E_A \subseteq V \times V$ is the set of edges of ancestrality;
- ➥ $r \in V$ is the root of the *DLDS*;
- ➥ $l : V \to \Gamma$ is a function, such that, for every $v \in V$, $l(v)$ is the (formula) label of $v$;
- ➥ $L : \bigcup_{i\in\mathscr{O}^0_\Gamma} E^i_D \to \mathscr{B}(\mathscr{O}_S)$ is a function, such that, for every $\langle u, v\rangle \in E^i_D$, $L(\langle u, v\rangle)$ is the bitstring representing from which formulas the $i$-th colored deduction edge $\langle u, v\rangle$ carries its dependency;
- ➥ $P : E_A \to \{1, \ldots, || \Gamma ||\}^\star$, such that, for every $e \in E_A$, $P(e)$ is a string of the form $o_1; \ldots; o_n$, where each $o_i$, $i = 1, \ldots, n$, is an ordinal in $\mathscr{O}_\Gamma$;

For each $i \in \mathscr{O}^0_\Gamma$, and, $\langle u, v\rangle \in E^i_D$, $i$ is called the color of the edge $\langle u, v\rangle$. Each deduction edge is colored with formulas from $\Gamma$ or the 0 color. The colors are introduced every time a collapsing of nodes as explained in [4] is performed. Tree-like greedy derivations have only 0 colored deduction edges. *DLDS*s obtained from Tree-like greedy derivations by effective collapsing of vertexes, sometime edges, have colored deduction edges. Obviously, not all *DLDS* is in the image of the function that maps Tree-like derivations into *DLDS*. It is also interesting to note that every decorated greedy tree-like derivation[3] is a

[2] Such that $n > 0$, for every $n \in \mathscr{O}_\Gamma$.

[3] Greedy tree-like derivations are defined in [4] too.

*DLDS* having $E_A$ empty and only deductive edges $E_D^i$ for $i = 0$, i.e., has only 0-colored deductive edges. In fact, any Natural Deduction (usually tree-like) derivation of a formula $\alpha$ can be seen as a *DLDS*, as shown in [4][4].

## 2.2 The Horizontal Compression Algorithm and Rules

In this article we try to provide the overall idea of the horizontal compression algorithm **HC**. The horizontal compression mentioned in the name is composed of a series of horizontal collapses. A horizontal collapse applies to a dag-like decorated greedy derivation. It aims to identify two or more nodes in the rooted dag-like derivation at the same deduction level. The collapsing applies from the conclusion level, namely the zero level, towards the assumptions levels. When applied to tree-like rooted and decorated derivations, it yields dags instead of trees. The following algorithm formally defines this operation as a case analysis, comprised of 26 rules and that applies to a dag-like derivation to yield a (new) dag-like derivation. The horizontal collapsing initially transforms tree-like derivations into dag-like derivations. Additional structure is needed to allow us to verify that a particular dag-like derivation is a (correct) derivation, indeed. We define a dag-like derivability structure as the underlying structure to encode dag-like derivations. Thus, a dag-like derivation is a *DLDS* instance, as defined in the above definition, and a condition that should be true about this *DLDS* instance. Below we can see the horizontal compression in its algorithmic form:

**Algorithm 2** Horizontal Compression

**Require:** A tree-like greedy derivation $\mathscr{D}$
**Ensure:** That the *DLDS* is $\mathscr{D}$ compressed

```
for l from 1 to h(𝒟) do
    for u and v at l do
        HCom(u, v)
    end for
end for
```

In this subsection, we show 9 of the 26 rules that define $HCom(u,v)$[5]. Out of these, the rules shown at Figure 5, Figure 6, and Figure 9 are new additions of this article (they are not described in [4]). Each rule applies to a specific pair of nodes (vertexes) of a *DLDS* $\mathscr{D}$, depicted at the left-hand side. The effect of collapsing these two nodes (vertexes) produces a new *DLDS*, depicted at the right-hand side of the rule. It is worth noting that every decorated greedy tree-like derivation is a *DLDS* having $P(v) = \varepsilon$, for every $v \in V$. These rules represent:

1. A collapse between two non-collapsed nodes:
   - 1.1. **Rule 1:** The conclusion of an $\supset$-Intro rule with the conclusion of an $\supset$-Elim rule (Figure 1);
   - 1.2. **Rule 2:** A hypothesis with the conclusion of an $\supset$-Elim rule (Figure 2);
   - 1.3. **Rule 3:** The conclusion of an $\supset$-Intro rule with a hypothesis (Figure 3);
   - 1.4. **Rule 4:** Two hypotheses (Figure 4);
   - 1.5. **New Rule 1:** The conclusions of two $\supset$-Elim rules (Figure 5); and
   - 1.6. **New Rule 2:** The conclusions of two $\supset$-Intro rules (Figure 6).

[4] As already said, the soundness of a *DLDS* is argued in [4].
[5] These 9 rules shown were the ones we have formalised in Lean.

2. A collapse between an already collapsed node and a non-collapsed node:
   - **2.1. Rule 5:** An already collapsed node with the conclusion of an $\supset$-Intro rule (Figure 7);
   - **2.2. Rule 6:** An already collapsed node with a hypothesis (Figure 8); and
   - **2.3. New Rule 3:** An already collapsed node with the conclusion of an $\supset$-Elim rule (Figure 9).

A hypothesis must be either an assumption or a top-formula. The symmetrical cases of rules **Rule 1**, **Rule 2**, and **Rule 3** are omitted without the loss of information.
Regarding the types of compression rules, let $\alpha$ and $\beta$ be formula occurrences in a derivation in $M_{\supset}$. For every formula occurrence in a derivation in $M_{\supset}$, either they are a hypothesis, or the conclusion of an $\supset$-Intro rule, or the conclusion of an $\supset$-Elim rule. Considering these possibilities, discounting symmetry, 6 is the total number of possible pairs of $\alpha$ and $\beta$. These 6 rules are what we call type-0 rules. We consider the full subgraph of the *DLDS*, to which the compression rules apply, to be determined by the set of nodes reachable from the nodes in their respective left-hand graph, which do not have any collapsed nodes. For example in **Rule 1**, represented at Figure 1, any node below the two bullets that are reachable from some of them is not a collapsed node.
There are also rules of type-1, which have as a precondition that their respective left-hand side graph represents a pair of nodes to collapse, such that exactly one of them is already the result of a previous collapse from either a type-0 or a type-1 rule. As stated in 2, the collapses follow the algorithm from the bottom up and left to right. There are three possible type-1 rules; depending on which rule, $\supset$-Intro or $\supset$-Elim, the right node to collapse is the conclusion of or if it is a hypothesis. All rules of type-0 and type-1, and only rules of type-0 and type-1, are shown in this article.
The rules of type-2, in contrast with type-0 and type-1, have as a precondition that the nodes to be collapsed are both targets of ancestor edges, i.e. members of $E_A$. What we stated is equivalent to saying that their respective sons collapsed according to the order of execution of algorithm 2.
Each item in the definition of a valid **DLDS** is an invariant property preserved by the application of all compression rules. In [4], the authors use this fact to prove Theorem 11, which states that the set of compression rules preserves validity of any **DLDS**.

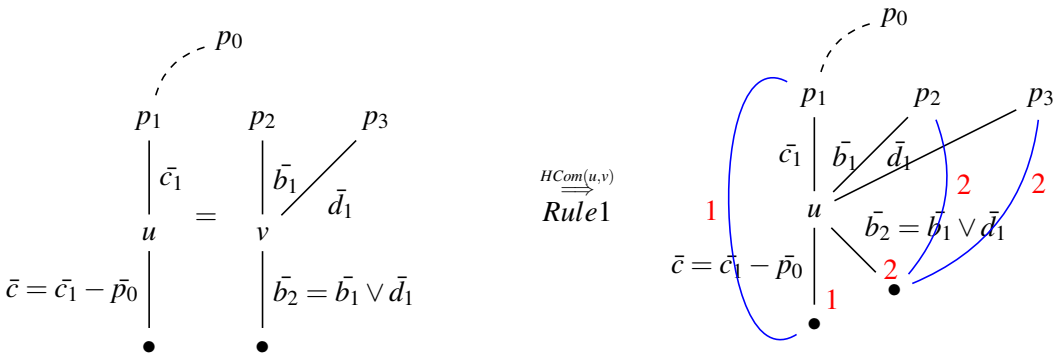



Figure 1: (a) Before collapse (b) After collapse $HCom(u,v)$

Figure 1 shows the first rule, named **Rule 1**. We use this figure to show how to read the pictorial representation of each horizontal compression rule. Both the left and the right-hand sides are subgraphs. In the left-hand side of the rule in Figure 1, $p_i$, $i = 1,3$, $u$ and $v$ are different nodes in the subgraph, such that $l(v) = l(u)$. The deductive edges are in black and have as labels the bit-string representing the dependency set denoted by $L$. For example, $L(\langle p_1,u\rangle) = \bar{c}_1$ shows that the deductive edge $\langle p_1,u\rangle \in E_d^0$

is labeled by the dependency set $Sets(\bar{c}_1)$. The absence of a label on an edge indicates that the edge is unlabeled. A label's node is $\bullet$ whenever it is not relevant what is its label to read the rule. Edges that belong to $E_D^i$ have the colour $i$; this is the red ordinal number $1,\ldots,n$ on a black deduction edge. The members of $E_A$, the ancestor edges, are coloured blue, and their labels under $P$ labelling function are red in the picture. For example, $\langle \bullet, p_1 \rangle \in E_A$ and $P(\langle \bullet, p_1 \rangle) = 1$. Moreover, we have that $\langle u, \bullet \rangle \in E_D^1$ in the graph in the right-hand side of **Rule 1**.

We advise the reader that in all graphical representations of the rules, both in this article and in [4], we do assume that nodes and edges drawn in different positions are always different. For example, in Figure 1 (i.e **Rule 1**), $p_0$, $p_1$, $p_2$, $p_3$, $u$, $v$ and the two bullets ($\bullet$) below them are all pairwise different nodes. Dashed lines represent paths in the graph.

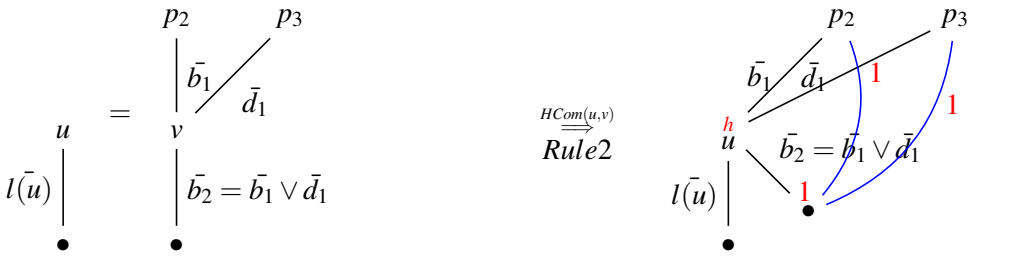



Figure 2: (a) Before collapse (b) After collapse $HCom(u,v)$

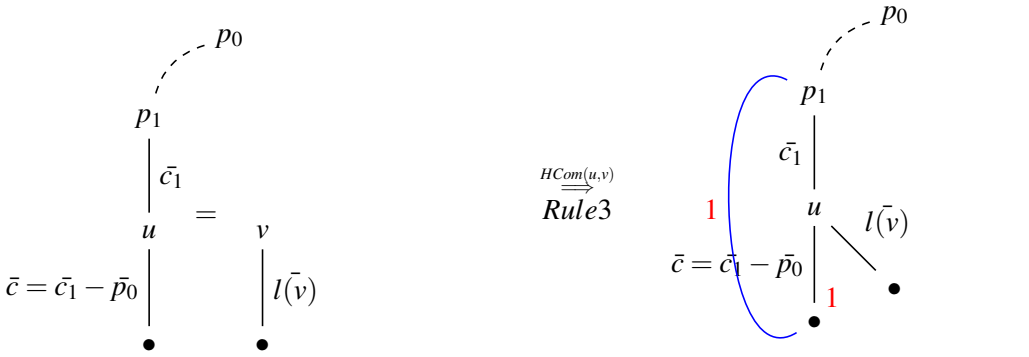



Figure 3: (a) Before collapse (b) After collapse $HCom(u,v)$


Figure 4: (a) Before collapse (b) After collapse $HCom(u,v)$

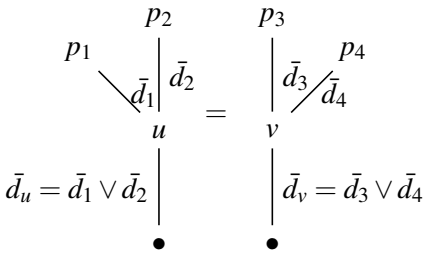
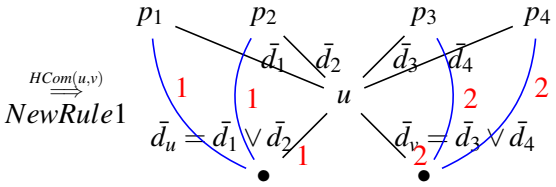



Figure 5: (a) Before collapse




(b) After collapse $HCom(u,v)$

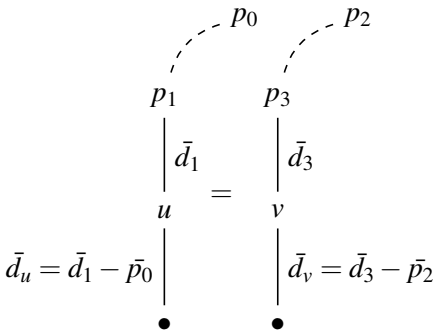
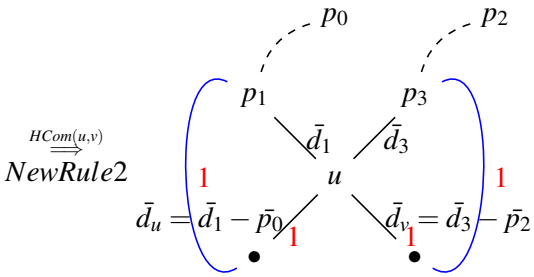



Figure 6: (a) Before collapse




(b) After collapse $HCom(u,v)$

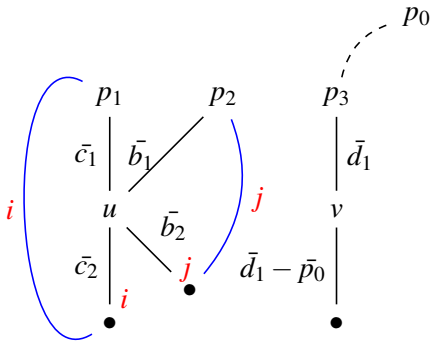
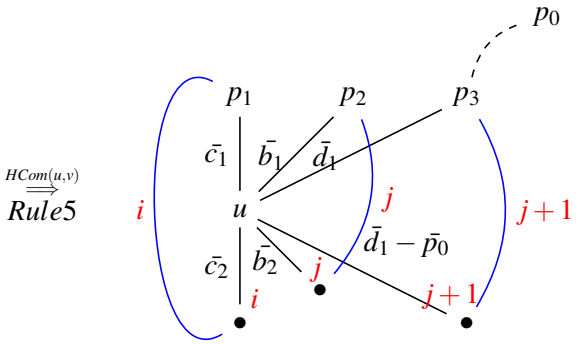



Figure 7: (a) Before collapse




(b) After collapse $HCom(u,v)$

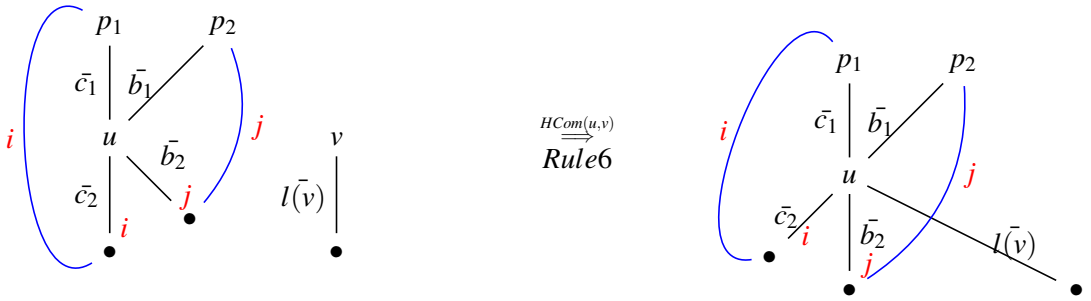



Figure 8: (a) Before collapse (b) After collapse $HCom(u,v)$

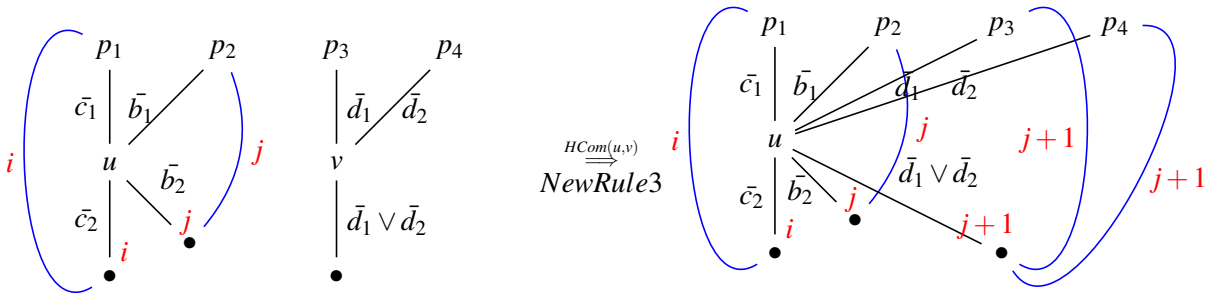



Figure 9: (a) Before collapse (b) After collapse $HCom(u,v)$

## 2.3 The Preservation of Soundness of the Compression Rules

Some further definitions concerning the compression rules and how they affect a *DLDS* must be made. All of these definitions were taken from [4], and we write them here in order to have a more self-contained document.

- ➤ **Definition 4 (Incoming Deductive Edges of a node):** Given a *DLDS* $\mathscr{D}$ of $\alpha$ from $\Gamma$ and a node $k \in \mathscr{D}$, the deductive in-degree of $k$ is defined as $INS(k) = \{f : f \in E_D^i, i \in \mathscr{O}(\Gamma \cup \{\alpha\})^0 \wedge target(f) = k\}$.
- ➤ **Definition 5 (Outgoing Deductive Edges from a node):** Given a *DLDS* $\mathscr{D}$ of $\alpha$ from $\Gamma$ and a node $k \in \mathscr{D}$, the deductive out-degree of $k$ is defined as $OUTS(k) = \{f : f \in E_D^i, i \in \mathscr{O}(\Gamma \cup \{\alpha\})^0 \wedge source(f) = k\}$.

Note that for any node $k$, both sets, $INS(k)$ and $OUTS(k)$, do not take into account the ancestor edges in their definition. We remember that the members of $E_A$ are not deductive edges. However, they play an important, though auxiliary role in the logical reading of any *DLDS*. A simple observation is that there is a natural map from a **D**ecorated **G**reed **T**ree-Like **D**erivation (*DGTD*) to a *DLDS*.

- ➤ **Definition 6:** Let $\mathscr{T} = \langle V, E_D, E_d, r, l, L \rangle$ be a *DGTD*. Let $\mathscr{O}_S$ be the order on the range of $l$, provided by $\mathscr{T}$ itself and $\Gamma$ the set of leaves in $\mathscr{T}$. Let $Dag(\mathscr{T})$ be $\langle V, (E_D^i)_{i \in \mathscr{O}_\Gamma^i}, E_A, r, l, L, P \rangle$, where $E_D^0 = E_D$, $E_D^i = \emptyset$, for all $i \neq 0$ and $E_A = \emptyset$, $P = \emptyset$.

It is easy to verify that $Dag(\mathscr{T})$ is well-defined, and hence it is a *DLDS*, for every *DGTD* $\mathscr{T}$. Thus, we have the mapping *Dag* from a *DGTD* to a *DLDS*. When reading a *DGTD* from top to bottom in a tree-like Natural Deduction derivation, there is at most one path from any top-formula occurrence to any other formula occurrence in the derivation. The following fact is an easy consequence of *Dag*'s definition above.

➤ **Proposition 1:** Let $\mathscr{T}$ be a *DGTD*. For every pair of nodes $v$ and $u$ in $\mathscr{T}$, there is a bijection between the paths from $v$ to $u$, in $\mathscr{T}$, and 0-paths, i.e., using only members of $E_D^0$, from $v$ to $u$ in $Dag(\mathscr{T})$. Moreover, the dependency sets, assigned by $L$, in both structures, are equal for every edge $\langle v,u\rangle \in E_D^0$.

From the definition of the mapping *Dag*, we can see that there is no path in the *DLDS* $Dag(\mathscr{T})$ with colors different from 0, due to $E_A^i = \emptyset$, for all $i \neq 0$. Moreover, there are no paths in $E_A$, for $E_A = \emptyset$.
The following definition shows how the information stored in the component $P$, the seventh one, the last, of any *DLDS* is used as a relative address for nodes in it. It uses:

➤ **Definition 7:** $el(\{e\}) = e$ and $el(S) = \perp$, if $S$ is not $\{a\}$ for some $a$.

➤ **Definition 8 (Relative Address of a Node):** Let $\mathscr{D}$ be a *DLDS* of $\alpha$ from $\Gamma$ and let $\gamma \in \mathscr{O}(\Gamma \cup \{alpha\})^0)^\star$ we say that $\gamma$ is the address of a node $v \in \mathscr{D}$ relative to a node $u \in \mathscr{D}$ iff the following algorithm 3 returns $v$ on input $\gamma$, $\mathscr{D}$ and $u$. The underlying idea is that $\gamma$ provides information on every branching in the path from $u$ downwards $v$. Each ordinal from left to right in $\gamma$ indicates which branch to take in.

**Algorithm 3** Finding a Node from its Relative Address and Origin of the Path

**Require:** $u$, the origin, $\mathscr{D}$, the *DLDS*, and the relative address $\gamma$

$b \leftarrow u$
$glues \leftarrow \gamma$
**while** $glues \neq \varepsilon$ **do**
  **if** $size(OUTS(b)) == 1$ **then**
    $g \leftarrow el(OUTS(b))$
    $b \leftarrow target(g)$
  **else if** $size(OUTS(b)) > 1 \wedge size(\{e/(e \in OUTS(b)) \wedge (color(e) = head(\gamma))\}) = 1$ **then**
    $g \leftarrow el(\{e/(e \in OUTS(b)) \wedge (color(e) = head(\gamma))\})$
    $b \leftarrow target(g)$
    $glues \leftarrow rest(\gamma)$
  **else**
    Return false
  **end if**
**end while**
Return b

For defining when a *DLDS* corresponds to a valid derivation we need the definition of Deductive path below.

➤ **Definition 9 (Deductive Path):** Given two nodes $v_1$ and $v_2$ in a **VLDS** $\mathscr{D} = \langle V, (E_D^i)_{i\in\{\bar{\lambda}\}\cup\mathscr{O}_\Gamma^i}, E_A, r, l, L, P\rangle$, we call a path $e_1, e_2, \ldots, e_n$ from $v_1$ to $v_2$ a deductive path, iff, for each $p = 1, \ldots, n$, $e_p \in \bigcup_{i\in\{\bar{\lambda}\}\cup\mathscr{O}_\Gamma^i} E_D^i$. In particular, if $e_1, e_2, \ldots, e_n$ is a deductive path from $v_1$ to $v_2$ and there is $i \neq 0$, such that $e_j \in E_D^i$ or $e_j \in E_D^{\bar{\lambda}}$, for some $0 \leq j \leq n$, then the path is a mixed deductive path from $v_1$ to $v_2$.

Given a *DLDS* $\mathscr{D} = \langle V, (E_D^i)_{i \in \mathscr{O}_\Gamma^i}, E_A, r, l, L, P \rangle$ and a node $w \in V$, we define:

1. **Pre(w) =** $\{v$ : Such that there is a deductive path from $v$ to $w\}$, as the set of nodes that are linked to $w$ by some deductive path;
2. **Top(w) =** $\{v$ : Such that $v \in Pre(w)$ and either $v$ is marked as hypothesis, or there is no $v' \in V$, or $\langle v', v \rangle \in (E_D^i)_{i \in \mathscr{O}_\Gamma^i}\}$, as the set of top nodes of a *DLDS*;
3. **DedPaths(w) =** $\{\langle e_1, \ldots, e_n \rangle$ : Such that $e_1 \ldots e_n$ is a deductive path, with $source(e_1) \in TopNode(w)$ and $target(e_n) = w\}$, as the set of full deductive paths reaching to $w \in V$.

➤ **Definition 10 (Relation $\sim$ between Dependency Sets):** For any pair of dependency sets $\bar{b}$ and $\bar{c}$, $\bar{b} \sim \bar{c}$ holds, if and only if, $\bar{c} = \bar{b}$ or $\bar{c} = \lambda$ or $\bar{b} = \lambda$.

➤ **Definition 11 (Flow):** Given a *DLDS* $\mathscr{D} = \langle V, (E_D^i)_{i\in\mathscr{O}_\Gamma^i}, E_A, r, l, L, P\rangle$ and a node $w \in V$, we define $Flow(\mathscr{D}, w)$ as a function from $Pre(w)$ into $\wp((\mathscr{O}_\Gamma^0)^* \times \mathscr{B}(\mathscr{O}_S))$, such that:

$Flow(\mathscr{D}, w)(v) =$

$$
\begin{cases}
\{(b(\vec{l}(v)), P(\langle v', v\rangle)) : \langle v', v\rangle \in E_A,\ v' \in V\} & \text{if } v \in Top(w), \text{ there is } v' \in V,\ \langle v', v\rangle \in E_A, \\
\{(b(\vec{l}(v)), \emptyset)\} & \text{if } v \in Top(w) \text{ and } \nexists v' \in V,\ \langle v', v\rangle \in E_A \\
\left\{ (\vec{b_1} \vee \vec{b_2}, p) : \boxed{\begin{array}{l} (v_1, v_2, v) \in \supset_E \text{ and} \\ (b_i, [o_i|p]) \in Flow(\mathscr{D}, w)(v_i), \\ \text{and } \langle v_i, v\rangle \in E_D^{o_i}, \text{ and,} \\ b_i \sim L(\langle v_i, v\rangle),\ i = 1, 2, \\ \text{OR } (v_1, v_2, v) \in \supset_E \text{ and} \\ (b_i, [0|p]) \in Flow(\mathscr{D}, w)(v_i),\ i = 1 \text{ or} \\ i = 2, \text{ and} \\ (b_j, \emptyset) \in Flow(\mathscr{D}, w)(v_j),\ j \neq i, \text{ and} \\ \langle v_i, v\rangle \in E_D^0,\ \langle v_j, v\rangle \in E_D^0, \text{ and} \\ b_i \sim L(\langle v_k, v\rangle),\ k = 1, 2, \end{array}} \right\} & \\
\cup & \\
\left\{ (\vec{b'} - \vec{\alpha}, p) : \boxed{\begin{array}{l} (v', v) \in \supset_I \text{ and} \\ (b', [o'|p]) \in Flow(\mathscr{D}, w)(v') \text{ and} \\ \vec{b'} \sim L(\langle v', v\rangle) \text{ and,} \\ \langle v', v\rangle \in E_D^{o'}, \text{ and } l(v) = \text{“}\alpha \supset l(v')\text{”} \end{array}} \right\} & \\
\cup & \\
\left\{ (b(\vec{l}(v)), \emptyset) : v \text{ is marked with } \hbar \text{ and } \nexists v',\ \langle v', v\rangle \in E_A \right\} & \text{otherwise} \\
\cup & \\
\left\{ (b(\vec{l}(v)), P(\langle v', v\rangle)) : v \text{ is marked with } \hbar \text{ and } \langle v', v\rangle \in E_A \right\} & \\
\cup & \\
\left\{ (\vec{b'} - \vec{\alpha}, P(\langle v_a, v\rangle)) : \boxed{\begin{array}{l} (v', v) \in \supset_I \text{ and} \\ (b', \emptyset) \in Flow(\mathscr{D}, w)(v') \text{ and} \\ v_a \in V,\ \langle v_a, v\rangle \in E_A \text{ and,} \\ \vec{b'} \sim L(\langle v', v\rangle), \text{ and} \\ l(v) = \text{“}\alpha \supset l(v')\text{”}, \text{ and} \\ \langle v', v\rangle \in E_D^0 \end{array}} \right\} & \\
\cup & \\
\left\{ (\vec{b_1} \vee \vec{b_2}, P(\langle v_a, v\rangle)) : \boxed{\begin{array}{l} (v_1, v_2, v) \in \supset_E \text{ and} \\ (b_i, \emptyset) \in Flow(\mathscr{D}, w)(v_i) \\ \text{and } v_a \in V,\ \langle v_a, v\rangle \in E_A, \text{ and} \\ \langle v_i, v\rangle \in E_D^0\ i = 1, 2, \text{ and} \\ b_k \sim L(\langle v_k, v\rangle),\ k = 1, 2 \end{array}} \right\} &
\end{cases}
$$

➤ **Definition 12:** Given a structure $\mathscr{D} = \langle V, (E_D^i)_{i\in\mathscr{O}_\Gamma^i}, E_A, r, l, L, P\rangle$, we say that it is a valid *DLDS*, iff, the following conditions hold on it:

- ➡ **Color-Acyclicity** For each $i \in \mathscr{O}_\Gamma^i$, $E_D^i$ does not have cycles;
- ➡ **Color-Leveled** The rooted sub-dag $\langle V, (E_D^i)_{i\in\mathscr{O}_\Gamma^i}, r\rangle$ is leveled;
- ➡ **Ancestor Edges** For each $\langle v_1, v_2\rangle \in E_A$, the level of $v_1$ is smaller than the level of $v_2$;
- ➡ **Ancestor Backway Information** For each $\langle v_1, v_2\rangle \in E_A$, $P(\langle v_1, v_2\rangle)$ is the relative address of $v_1$ from $v_2$;
- ➡ **Simplicity** The rooted sub-dag $\langle V, (E_D^i)_{i\in\mathscr{O}_\Gamma^i}, r\rangle$ is a simple graph, i.e, for each pair of nodes $v_1$ and $v_2$, there is at most an $i \in \mathscr{O}_\Gamma^i$, such that $\langle v_1, v_2\rangle \in E_D^i$;
- ➡ **Ancestor-Simplicity** The sub-dag $\langle V, E_A\rangle$ is a simple graph;
- ➡ **Non-Nested Ancestor Edges** For each $\langle v_1, v_2\rangle \in E_A$, there is no $w$ in the path from $v_2$ to $v_1$, determined by $P(\langle u, v\rangle \in E_A)$, such that $\langle w, z\rangle \in E_A$, for some $z \in E_A$;
- ➡ **Correct Rule Application** For each $w \in V$, $Flow(\mathscr{D}, w)(v)$ is well-defined for each $v \in Pre(w)$. Moreover, for each $w$ and $v$, $Flow(\mathscr{D}, w)(v)$, with $v \in Pre(w)$, we have:
  - If $Flow(\mathscr{D}, w)(v) = \{(\vec{b}, p)\}$ then $OUT(v) = \{\langle v, v'\rangle\}$ and the color of $\langle v, v'\rangle$ is $head(p)$, i.e., $\langle v, v'\rangle \in E_D^{head(p)}$, and $\vec{b} = L(\langle v, v'\rangle)$, and;
  - If $Flow(\mathscr{D}, w)(v) \neq \emptyset$ and it is not a singleton either then for each $\Phi_i = \{(\vec{b}, p) \in Flow(\mathscr{D}, w)(v) : head(p) = i\}$:
    - If $\Phi_i \neq \emptyset$ then there is only one $v'$ $\langle v, v'\rangle \in E_D^i$ and if $\Phi_i = \{(\vec{b}, p)\}$ then $L(\langle v, v'\rangle) = \vec{b}$ else $L(\langle v, v'\rangle) = \lambda$, and;
    - If $\Phi_i = \emptyset$ then there is no $v' \in V$, such that, $\langle v, v'\rangle \in E_D^i$.

Each of the items in definition 2.3 is an invariance property that should be preserved by all compression rules applications. It is worth noting that in **Correct Rule Application**, the verification that a rule application is correct involves, among other things, finding out that the premises agree with the conclusion and checking that the dependency sets are correctly assigned, this is the main role of function $Flow$.

# 3 Formalization in Lean

This section describes the beginnings of a formalization in Lean (Lean 3, v0.16.53) for the following theorem (Theorem 12 in [4]):

➤ **Main Theorem:**
If the algorithm **HC** is applied to a valid *DLDS*, then it eventually halts exiting a *DLDS* that has no level with two nodes labeled with the same formula.

The more complex/necessary proofs, lemmas, and definitions have been prioritized over simpler ones in this section. To the former, we provide full explanations and their formalization in Lean. To the latter, we provide only a brief explanation, and their formalization is omitted. The complete formalization in Lean can be viewed at [5].

## 3.1 Type Definitions

Because our result is about the compression rules that define $HCom(u, v)$, we need to create a type for the entries $u$ and $v$. In this context, $u$ and $v$ indicate the nodes to be collapsed by the application of $HCom(u, v)$. However, the information of which nodes $u$ and $v$ represent is insufficient for our proof.

Not only the nodes to be collapsed, we need to know an entire neighborhood of arrows and vertexes around the nodes $u$ and $v$, therefore:

```
inductive neighborhood
| dag (CENTER : node)
      (IN : list deduction)
      (OUT : list deduction)
      (ANCESTRAL : list ancestral)
export neighborhood (dag)
```

The `neighborhood` is directly defined by the compression rules. Every instance of the `neighborhood` type is composed of 4 distinct parts: a central node which is used for the collapse (CENTER); a list of deduction edges arriving at that central node (IN); a list of deduction edges exiting from that central node (OUT); and a list of edges of ancestrality pertinent to the compression rule (ANCESTRAL). These parts are given as parameter to the `neighborhood` type constructor. The types of these parameters, `node`, `deduction`, and `ancestral`, are defined as follows:

```
inductive node
| vertex (LEVEL : ℕ)
         (LABEL : ℕ)
         (FORMULA : formula)
export node (vertex)

inductive deduction
| arrow (START : node)
        (END : node)
        (COLOUR : list ℕ)
        (DEPENDENT : set formula)
export deduction (arrow)

inductive ancestral
| path (START : node)
       (END : node)
       (PATH : list ℕ)
export ancestral (path)
```

In the `node` type's definition, a node's level and label (an identifier unique to that node) are each represented by a natural number, which must be given as parameters to the type constructor. This representation is justifiable because the number of possible levels/labels in a *DLDS* is always a natural number, so any arbitrary ordering over the set of possible levels/labels creates a bijection between the set of levels/labels and the natural numbers. The level parameter of a node is used to associate nodes of the *DLDS* for collapse, and check if they are at the same level of the *DLDS*. Using a parameter to represent the label of a node makes it possible to differentiate any two nodes of the tree even when looked at in isolation. These parameters allow for a better, more precise categorization of the nodes, something which our proof requires. The last part of a node is its formula, which in this context is a $M_{\supset}$ formula. The `formula` type is defined as follows:

```
inductive formula
| atom (SYMBOL : ℕ) : formula
| implication (ANTECEDENT CONSEQUENT : formula) : formula
export formula (atom implication)
notation #SYMBOL := formula.atom SYMBOL
notation ANTECEDENT >> CONSEQUENT := formula.implication ANTECEDENT CONSEQUENT
```

The `deduction` type is used to instantiate a neighborhood's deduction edges while the `ancestral` type is used to instantiate a neighborhood's edges of ancestrality. A deduction edge is composed of: a starting node (START); an end node (END); an identifying colour (COLOUR); and a dependency set (DEPENDENT). An edge of ancestrality is composed of: a starting node (START); an end node (END); an identifying colour path (PATH). The formalisation of both these types is taken as a direct translation from their definition as stated in the previous sections.

### 3.2 Proving the Main Theorem

We are still working on our Lean-assisted proof for the ➤ **The Main Theorem**. As of the writing of this article, we've achieved a partial result, at `Main_Lemma`, showing that:

➤ **Main Lemma:**
Let $HCom(u,v)$ be the application of either a type-0 or a type-1 compression rule to nodes $u$ and $v$ of a *DLDS* $\mathscr{D}$. Let $u'$ be the resulting collapsed node and $\mathscr{D}'$ the resulting *DLDS* after the collapse. Let $v'$ be a node in $\mathscr{D}'$, at the same level and with the same $M_{\supset}$ formula as $u'$. Then $HCom(u',v')$ can only be the application a type-1 compression rule.

Central to the Lean-assisted proof of the lemma above are the methods: `case_neighborhood_01`, which validates if a given neighborhood of the *DLDS* represents a node that is the conclusion of an application of $\supset$-Elim; `case_neighborhood_02`, which validates if a given neighborhood of the *DLDS* represents a node that is the conclusion of an application of $\supset$-Intro; `case_neighborhood_03`, which validates if a given neighborhood of the *DLDS* represents a node that is a hypothesis; `case_neighborhood_04`, which validates if a given neighborhood of the *DLDS* represents a node that is the resulting collapsed node of an application of either a type-0 or a type-1 compression rule; and `horizontal_collapse`, which takes two applicable neighborhoods and exits a collapsed neighborhood, much like the figures shown on the previous Subsection 2.2. By analysing these four categories of neighborhoods, and considering the method for horizontal collapse, the proof of the lemma is done by the comparison of each of the 9 cases, again disregarding symmetrical cases.
We wrote the formalization to more closely resemble a computer program, motivated by the reasoning that this would make it easier for us to write it and later for the reader to comprehend it. Usage of Lean's tactics mode, marked by keywords `begin` and `end`, means that, when processing the input bracketed by the keywords, the theorem prover can execute each tactic in a compound sequence to produce an expression of its required type. It also helps the user keep track of the multiple goals and subgoals involved in a proof by forcing the theorem prover to show that kind of information. On the topic of goals/subgoals, tactics `have` and `from` are used to introduce new subgoals and solve existing goals/subgoals, respectively. The tactic `from` provides an exact proof term as the solution, meaning that if `G` is a goal and `t` is a term of type `T`, then `from t` succeeds, iff, `G` and `T` can be unified. Finally, `assume` is used here to introduce and name all our hypotheses and variables.

## 4 Formal Proofs and Interactive Theorem Proving in Lean

Though our Lean-assisted proof was based on a proof in [4], hurdles had to be overcome during formalization, even at this initial work. The most problematic to tackle was choosing the native types to be used, like the $\mathbb{N}$, `list`, and `set` types, and defining our own, like the inductive `formula`, `node`, `deduction`, `ancestral`, and `neighborhood` types. More than once we needed to revise these choices and definitions, upon noticing something missing or an inconsistency in our formalisation.
Still, the interactive theorem prover had many positive points in its favor. The proof and definitions could be divided into modules, which could be tackled one at a time. Lean's implementation of the `sorry` keyword, which can substitute any tactic or proof term, regardless of context, was also immensely beneficial, allowing us to postpone some parts of the formalization without hindering its progress. Removing some of those `sorry` keywords was the last thing we did before finishing the proof. Using Lean in iterative mode, accessed by writing code between `begin` and `end`, not only was helpful to us when writing the formalization, but we also believe it aids the reader in understanding it. In iterative mode, the theorem prover highlights the current goal after the use of each tactic and also keeps track of variables and propositions currently in context. Though not all of it, Lean already had several types, proofs, and other such definitions we needed already implemented in one of its many libraries. Finally, unlike pen-and-paper proofs, our proof in Lean is an object that can be manipulated and verified, preferably with the assistance of the prover itself, which should hopefully make it more credible and accessible.

## 5 Conclusion

This paper is a continuation of the work in [4]. Here we present the initial step towards of a Lean-assisted proof showing that **HC** halts for every $M_{\supset}$ tautology, exiting a valid *DLDS* with no two equal nodes on the same level of the *DLDS*. This result is called Theorem 12 in [4], where the authors show a less thorough proof of this result.
During the formalization in Lean, finding succinct concepts and definitions to define a *DLDS* was a particular challenge. Like a Natural Deduction tree, a *DLDS* has levels over which recursion can be applied. An argument can be made that this characteristic is due to the visual nature of the tree type, of which a *DLDS* is based, which is not intuitive when defining the type in Lean. However, we realised that there is no need to ever instantiate an entire *DLDS* to prove our result. This allowed us to work with the `neighborhood` type, which has a much more direct definition based on the compression rules, instead of a *DLDS* type.
Using the already implemented $\mathbb{N}$, `list`, and `set` types required the formalization of some additional definitions. Though the number of lemmata proven using these methods was worthy of note, most of their proofs come directly from the in-built recursion over Lean's $\mathbb{N}$, `list`, and `set` types.
Experiments with the compression of both naive and huge proofs in natural deduction for the non-hamiltonianicity of some graphs, such as a Petersen graph, was done in [1]. A compression ratio of almost 90% was obtained, with the bigger and more redundant proofs having the best compression ratio after removing their redundant parts.
In the future, we will continue this work by finalising the Lean-assisted proof of ➤ **The Main Theorem**, by including the compression rules of type-2 to the formalisation. Later, we also intend on writing a Lean-assisted proof to show the soundness of each compression rule, or that the set of compression rules preserves validity of any **DLDS**, as stated in [4] (Theorem 11).

## References

[1] BARROS JÚNIOR, J.F.C.; HAEUSLER, E.H. (2019): ***A comparative study on compression techniques for Propositional Proofs***. In: *Book of Abstracts, 19th Braz. Meeting on Logic*, Brazil, pp. 85–86.

[2] DE MOURA, L.; KONG, S.; AVIGAD, J.: ***Theorem Proving in Lean.*** Available at https://leanprover.github.io/theorem_proving_in_lean/. Accessed: June 2022.

[3] DE MOURA, L.; KONG, S.; AVIGAD, J.; VAN DOORN, F.; VON RAUMER, J. (2015): ***The Lean theorem prover (system description)***. In: *Proceedings of the 25th International Conference on Automated Deduction*, Berlin, Germany, pp. 378–388.

[4] HAEUSLER, E.H.; BARROS JÚNIOR, J.F.C.: ***On the horizontal compression of dag-derivations in minimal purely implicational logic.*** Available at https://arxiv.org/pdf/2206.02300.pdf. Accessed: June 2022.

[5] MOURA BRASIL FILHO, R.C.: ***Horizontal-Compression***. Available at https://github.com/Robilsu/Horizontal-Compression. Accessed: June 2022.

[6] PRAWITZ, D. (1965): *Natural Deduction: Proof-Theoretical Study*. Dover Publications.

# A logic for paraconsistent transition systems

Ana Cruz
INESC TEC, University of Minho, Portugal

Alexandre Madeira
CIDMA, University of Aveiro, Portugal

Luís S. Barbosa
INESC TEC, University of Minho, Portugal

Often in Software Engineering a modelling formalism has to support scenarios of inconsistency in which several requirements either reinforce or contradict each other. Paraconsistent transition systems are proposed in this paper as one such formalism: states evolve through two accessibility relations capturing weighted evidence of a transition or its absence, respectively. Their weights come from a specific residuated lattice. A category of these systems, and the corresponding algebra, is defined providing a formal setting to model different application scenarios. One of them, dealing with the effect of quantum decoherence in quantum programs, is used for illustration purposes.

## 1 Introduction

Dealing with application scenarios where requirements either reinforce or contradict each other is not uncommon in Software Engineering. One such scenarios comes from current practice in quantum computation in the context of NISQ (*Noisy Intermediate-Scale Quantum*) technology [11] in which levels of decoherence of quantum memory need to be articulated with the length of the circuits to assess program quality.

In a recent paper [7], the authors introduced a new kind of weighted transitions systems which records, for each transition, a positive and negative weight which, informally, capture the degree of effectiveness (*'presence'*) and of impossibility (*'absence'*) of a transition. This allows the model to capture both *vagueness*, whenever both weights sum less than 1, as usual e.g. in fuzzy systems, and *inconsitency*, when their sum exceeds 1. This last feature motivates the qualifier *paraconsistent* borrowed from the work on paraconsistent logic [9, 5], which accommodates inconsistency in a controlled way, treating inconsistent information as potentially informative. Such logics were originally developed in Latin America in the decades of 1950 and 1960, mainly by F. Asenjo and Newton da Costa. Quickly, however, the topic attracted attention in the international community and the original scope of mathematical applications broadened out, as witnessed in a recent book emphasizing the engineering potential of paraconsistency [2]. In particular, a number of applications to themes from quantum mechanics and quantum information theory have been studied by D. Chiara [4] and W. Carnielli and his collaborators [1, 6].

This paper continues such a research program in two directions. First it introduces a suitable notion of morphism for paraconsistent labelled transition systems (PLTS) leading to the definition of the corresponding category and its algebra. Notions of simulation, bisimulation and trace for PLTS are also discussed. On a second direction, the paper discusses an application of PLTS to reason about the effect of quantum decoherence in quantum programs.

**Paper structure.** After recalling the concept of a PLTS and defining their morphisms in section 2, section 3 discusses suitable notions of simulation, bisimulation and trace. Compositional construction of

(pointed) PLTS are characterised in section 4 by exploring the relevant category, following G. Winskel and M. Nielsen's 'recipe' [13]. Section 5 illustrates their use to express quantum circuits with decoherence. Finally, section 6 concludes and points out a number of future research directions.

## 2 Paraconsistent labelled transition systems

A *paraconsistent labelled transition system* (PLTS) incorporates two accessibility relations, classified as positive and negative, respectively, which characterise each transition in opposite ways: one represents the evidence of its presence and other the evidence of its absence. Both relations are weighted by elements of a residuated lattice $\Sigma = \langle \wedge, \vee, \odot, \rightarrow, 1, 0 \rangle$, where, $\langle A, \wedge, \vee, 1, 0 \rangle$ is a lattice, $\langle A, \odot, 1 \rangle$ is a monoid, and operation $\odot$ is residuated, with $\rightarrow$, i.e. for all $a, b, c \in A$, $a \odot b \leq c \Leftrightarrow b \leq a \rightarrow c$. A Gödel algebra $G = \langle [0,1], min, max, min, \rightarrow, 0, 1 \rangle$ is an example of such a structure, that will be used in the sequel. Operators *max* and *min* retain the usual definitions, whereas implication is given by

$$a \rightarrow b = \begin{cases} 1, \text{ if } a \leq b \\ b, \text{ otherwise} \end{cases} .$$

Our constructions, however, are, to a large extent, independent of the particular residuated lattice chosen. The definition below extends the one in reference [7] to consider labels in an explicit way. Thus,

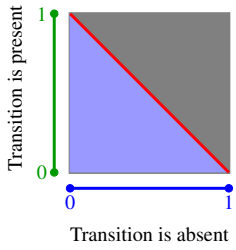
**Definition 1.** *A **paraconsistent labelled transition system** (PLTS) over a residuated lattice $A$ and a set of atomic actions $\Pi$ is a structure $\langle W, R, \Pi \rangle$ where, $W$ is a non-empty set of states, $\Pi$ is a set of labels, and $R \subseteq W \times \Pi \times W \times A \times A$ characterises its dynamics, subjected to the following condition: between two arbitrary states there is at most one transition involving label $a$, for every $a \in \Pi$. Each tuple $(w_1, a, w_2, \alpha, \beta) \in R$ represents a transition from $w_1$ to $w_2$ labelled by $(a, \alpha, \beta)$, where $\alpha$ is the degree to which the action $a$ contributes to a transition from $w_1$ to $w_2$, and $\beta$, dually, expresses the degree to which it prevents its occurrence.*

The condition imposed in the definition above makes it possible to express relation $R$ in terms of a *positive* and a *negative* accessibility relation $r^+, r^- : \Pi \longrightarrow A^{W \times W}$, with

$$r^+(\pi)(w, w') = \begin{cases} \alpha \text{ if } (w, \pi, w', \alpha, \beta) \in R \\ 0 \text{ otherwise} \end{cases}$$

and $r^-$ defined similarly. These two relations jointly express different behaviours associated to a transition:

- *inconsistency*, when the positive and negative weights are contradictory, i.e. they sum to some value greater then 1; this corresponds to the upper triangle in the picture below, filled in grey.
- *vagueness*, when the sum is less than 1, corresponding to the lower, periwinkle triangle in the same picture;
- *consistency*, when the sum is exactly 1, which means that the measures of the factors enforcing or preventing a transition are complementary, corresponding to the red line in the picture.

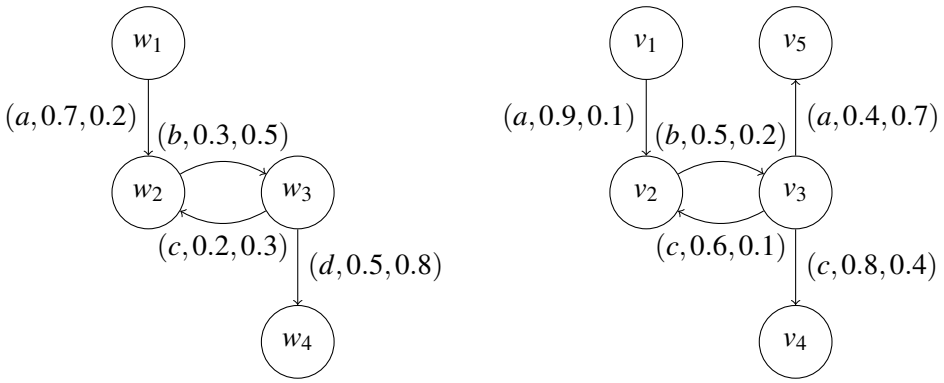


Morphisms between PLTS respect, as one would expect, the structure of both accessibility relations. Formally,

**Definition 2.** *Let $T_1 = \langle W_1, R_1, \Pi \rangle$, $T_2 = \langle W_2, R_2, \Pi \rangle$ be two PLTSs defined over the same set of actions $\Pi$. A **morphism** from $T_1$ to $T_2$ is a function $h : W_1 \to W_2$ such that*

$$\forall_{a \in \Pi},\ r_1^+(a)(w_1, w_2) \leq r_2^+(a)(hw_1, hw_2) \text{ and } r_1^-(a)(w_1, w_2) \geq r_2^-(a)(hw_1, hw_2)$$

**Example 1.** *Function $h = \{w_1 \mapsto v_1, w_2 \mapsto v_2, w_3 \mapsto v_3\}$ is a morphism from $M_1$ to $M_2$, over $\Pi = \{a, b, c, d\}$, depicted below*




## 3 Simulation and Bisimulation for PLTS

Clearly, PLTSs and their morphisms form a category, with composition and identities borrowed from Set. To compare PLTSs is also useful to define what simulation and bisimulation mean in this setting. Thus, under the same assumptions on $T_1$ and $T_2$,

**Definition 3.** *A relation $S \subseteq W_1 \times W_2$ is a **simulation** provided that, for all $\langle p, q \rangle \in S$ and $a \in \Pi$,*

$$p \xrightarrow{(a,\alpha,\beta)}_{T_1} p' \Rightarrow \langle \exists_{q' \in W_2} . \exists_{\gamma,\delta \in [0,1]} . \ q \xrightarrow{(a,\ \gamma,\ \delta)}_{T_2} q' \ \wedge \ \langle p', q' \rangle \in S \ \wedge \ \gamma \geq \alpha \ \wedge \ \delta \leq \beta \rangle$$

*which can be abbreviated to*

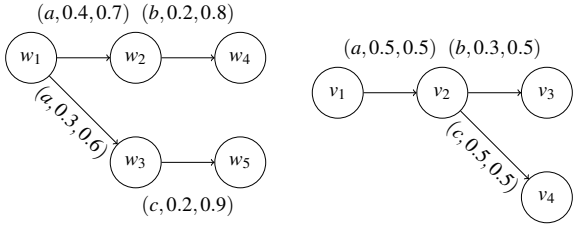
$$p \xrightarrow{(a,\alpha,\beta)}_{T_1} p' \Rightarrow \langle \exists_{q' \in W_2} . \ q \xrightarrow{(a,\ \gamma:\ \gamma \geq \alpha\ ,\ \delta:\ \delta \leq \beta)}_{T_2} q' \ \wedge \ \langle p', q' \rangle \in S \rangle$$

*Two states $p$ and $q$ are **similar**, written $p \lesssim q$, if there is a simulation $S$ such that $\langle p, q \rangle \in S$.*

Whenever one restricts in the definition above to the existence of values $\gamma$ (resp. $\delta$) such that $\gamma \geq \alpha$ (resp. $\delta \leq \beta$), the corresponding simulation is called *positive* (resp. *negative*).

**Example 2.** *In the PLTSs depicted below, $w_1 \lesssim v_1$, witnessed by*

$$S = \{\langle w_1, v_1 \rangle, \langle w_2, v_2 \rangle, \langle w_3, v_2 \rangle, \langle w_4, v_3 \rangle, \langle w_5, v_4 \rangle\}$$



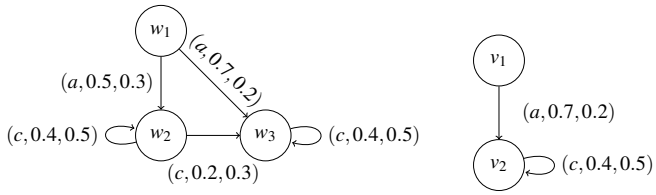
Finally,

**Definition 4.** *A relation $B \subseteq W_1 \times W_2$ is a **bisimulation** if for $\langle p,q\rangle \in B$ and $a \in \Pi$*

$$p \xrightarrow{(a,\alpha,\beta)}_{M_1} p' \Rightarrow \langle \exists q' \in W_2 : q \xrightarrow{(a,\alpha,\beta)}_{M_2} q' \wedge \langle p',q'\rangle \in B\rangle$$
$$q \xrightarrow{(a,\alpha,\beta)}_{M_2} q' \Rightarrow \langle \exists p' \in W_1 : p \xrightarrow{(a,\alpha,\beta)}_{M_1} p' \wedge \langle p',q'\rangle \in B\rangle$$

*Two states $p$ and $q$ are **bisimilar**, written $p \sim q$, if there is a bisimulation $B$ such that $\langle p,q\rangle \in B$.*

**Example 3.** *Consider the two PLTSs depicted below. Clearly, $w_1 \sim v_1$.*




**Lemma 1.** *Similarity, $\lesssim$, and bisimilarity, $\sim$, form a preorder and an equivalence relation, respectively.*

*Proof.* The proof is similar to one for classical labelled transition systems (details in [8]). □

As usual, a *trace* from a given state $w$ in a PLTS $T$ is simply the sequence $s$ of tuples $(a,\alpha,\beta)$ labelling a path in $T$ starting at $w$. A first projection on such a sequence, i.e. $\pi_1^*(s)$ retrieves the corresponding sequence of labels that constitutes what may be called an *unweighted trace*. More interesting is the notion of *weighted trace* which appends to the sequence of labels, the maximum value for the positive accessibility relation and the minimum value for the negative accessibility relation computed along the trace $s$. Formally,

**Definition 5.** *Given a trace $s$ in a PLTS $T$, the corresponding* weighted trace *is defined by*

$$tw(s) \;=\; \langle \pi_1^*, \bigwedge(\pi_2^*), \bigvee(\pi_3^*)\rangle\,(s)$$

*where, $\pi_n$ denotes the $n$ projection in a tuple, $\langle f,g,h\rangle$ is the universal arrow to a Cartesian product, $f^*$ is the functorial extension of $f$ to sequences over its domain, and $\bigwedge$ (resp. $\bigvee$) are the distributed version of $\wedge$ (resp. $\vee$) over sequences.*

**Definition 6.** *A weighted trace $t = \langle [a_1,a_2,...,a_m],\alpha,\beta\rangle$ is a **weighted subtrace** of $t' = \langle [b_1,b_2,...,b_n],\gamma,\delta\rangle$ if (i) sequence $[a_1,a_2,...,a_m]$ is a prefix of $[b_1,b_2,...,b_n]$, (ii) $\gamma \geq \alpha$ and (iii) $\delta \leq \beta$. The definition lifts to sets as follows: given two sets $X$ and $Y$ of weighted traces,*

$$X \sqsubseteq Y \;\; \textit{iff} \;\; \forall_{t\in X}.\exists_{t'\in Y}.\; t \textit{ is a weighted subtrace of } t'$$

**Example 4.** *Consider again the two PLTSs given in Example 2. The weighted traces from $w_1$ are $\{t_1 = \langle [a,b],0.2,0.8\rangle, t_2 = \langle [a,c],0.2,0.9\rangle\}$ and the ones from $v_1$ are $\{t_1' = \langle [a,b],0.5,0.5\rangle, t_2' = \langle [a,c],0.5,0.5\rangle\}$. Clearly, $t_1$ (resp. $t_2$) is a weighted subtrace of $t_1'$ (resp. $t_2'$).*

**Lemma 2.** *Consider two PLTSs, $T_1 = \langle W_1,R_1\rangle$ and $T_2 = \langle W_2,R_2\rangle$. If two states $p \in W_1$ and $q \in W_2$ are similar (resp. bisimilar), i.e., $p \lesssim q$ (resp. $p \sim q$), then the set of weighted traces from $p$, $X$, and the set of weighted traces from $q$, $Y$, are such that $X \sqsubseteq Y$ (resp. coincide).*

*Proof.* If $p \lesssim q$ each trace $t$ from $p$ is a prefix of trace $t'$ from $q$. Let $[\alpha_1, \alpha_2, ..., \alpha_m]$ and $[\beta_1, \beta_2, ..., \beta_m]$ be the sequences of positive and negative weights associated to $t$. Similarly, let $[\alpha'_1, \alpha'_2, ..., \alpha'_n]$ and $[\beta'_1, \beta'_2, ..., \beta'_n]$ be the corresponding sequences for $t'$; of course $m \leq n$. As $(p,q)$ belongs to a simulation, $\alpha'_i \geq \alpha_i$ and $\beta'_i \leq \beta_i$, for all $i \leq n$. So, $Min[\alpha'_1, \alpha'_2, ..., \alpha'_m] \geq Min[\alpha_1, \alpha_2, ..., \alpha_m]$ and $Max[\alpha'_1, \alpha'_2, ..., \alpha'_m] \leq Max[\alpha_1, \alpha_2, ..., \alpha_m]$. Note that $Min$ and $Max$ correspond to $\bigwedge$ and $\bigvee$ in a Gödel algebra. Thus,

$$\langle t, Min[\alpha_1, \alpha_2, ..., \alpha_n], Max[\alpha_1, \alpha_2, ..., \alpha_n] \rangle$$

is a weighted subtrace of $\langle t'|_m, Min[\alpha'_1, \alpha'_2, ..., \alpha'_n], Max[\alpha'_1, \alpha'_2, ..., \alpha'_n] \rangle$, where $t'|_m$ is the subsequence of $t$ with $m$ elements. The statement for $\sim$ follows similarly. □

Note that the converse of this lemma does not hold, as shown by the following counterexample.

**Example 5.** *Consider the PLTS depicted below.*


$X = \{\langle [a], 0.5, 0.3 \rangle, \langle [a,b], 0.5, 0.3 \rangle\}$ *is the set of weighted traces from* $w_1$. *Similarly,* $Y = \{\langle [a], 0.7, 0.2 \rangle, \langle [a,b], 0.5, 0.3 \rangle\}$ *is the corresponding set from* $w_2$. *Clearly* $\langle [a], 0.5, 0.3 \rangle$ *is a weighted subtrace of* $\langle [a], 0.7, 0.2 \rangle$. *Thus* $X \sqsubseteq Y$. *However,* $w_1 \not\lesssim w_2$.

# 4 New PLTS from old

New PLTS can be built compositionally. This section introduces the relevant operators by exploring the structure of the category of Pt of *pointed* PLTS, i.e. whose objects are PLTSs with a distinguished initial state, i.e. $\langle W, i, R, \Pi \rangle$, where $\langle W, R, \Pi \rangle$ is a PLTS and $i \in W$. Arrows in Pt are allowed between PLTSs with different sets of labels, therefore generalizing Definition 2 as follows:

**Definition 7.** *Let* $T_1 = \langle W_1, i_1, R_1, \Pi \rangle$ *and* $T_2 = \langle W_2, i_2, R_2, \Pi' \rangle$ *be two pointed PLTSs. A morphism in* Pt *from* $T_1$ *to* $T_2$ *is a pair of functions* $(\sigma : W_1 \to W_2,\ \lambda : \Pi \to_\perp \Pi'$*) such that*[1] $\sigma(i_1) = i_2$, *and, if* $(w, a, w', \alpha, \beta) \in R_1$ *then* $(\sigma(w), \lambda(a), \sigma(w'), \alpha', \beta') \in {R_2}^\perp$, *with* $\alpha \leq \alpha'$ *and* $\beta' \leq \beta$, *where, for an accessibility relation* $R$, $R^\perp = R \cup \{(w, \perp, w, 1, 0) \mid w \in W\}$ *denotes* $R$ *enriched with* idle *transitions in each state.*

Clearly Pt forms a category, with composition inherited from Set and $\mathsf{Set}_\perp$, the later standing for the category of sets and partial functions, with $T_{nil} = \langle \{*\}, *, \emptyset, \emptyset \rangle$ as both the initial and final object. The corresponding unique morphisms are $! : T \to T_{\text{nil}}$, given by $\langle \underline{*}, () \rangle$, and $? : T_{nil} \to T$, given by $\langle \underline{i}, () \rangle$, where $()$ is the empty map and notation $\underline{x}$ stands for the constant, everywhere $x$, function.

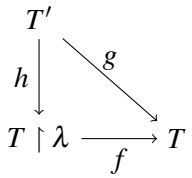
An algebra of PLTS typically includes some form of parallel composition, disjoint union, restriction, relabelling and prefixing, as one is used from the process algebra literature [3]. Accordingly, these operators are defined along the lines proposed by G. Winskel and M. Mielsen [13], for the standard, more usual case.

[1] Notation $\lambda : \Pi \to_\perp \Pi'$ stands for the totalization of a partial function by mapping to $\perp$ all elements of $\Pi$ for which the function is undefined.

**Restriction.** The restriction operator is intended to control the interface of a transition system, preserving, in the case of a PLTS, the corresponding positive and negative weights. Formally,

**Definition 8.** *Let $T = \langle W, i, R, \Pi \rangle$ be a PLTS, and $\lambda : \Pi' \to \Pi$ be an inclusion. The **restriction** of $T$ to $\lambda$, $T \restriction \lambda$, is a PLTS $\langle W, i, R', \Pi' \rangle$ over $\Pi'$ such that $R' = \{(w, \pi, w', \alpha, \beta) \in R \mid \pi \in \Pi'\}$.*

There is a morphism $f = (1_W, \lambda)$ from $T \restriction \lambda$ to $T$, and a functor $P : \mathsf{Pt} \to \mathsf{Set}_\perp$ which sends a morphism $(\sigma, \lambda) : T \to T'$ to the partial function $\lambda : \Pi' \to \Pi$. Clearly, $f$ is the Cartesian lifting of morphism $P(f) = \lambda$ in $\mathsf{Set}_\perp$. Being Cartesian means that for any $g : T' \to T$ in $\mathsf{Pt}$ such that $P(g) = \lambda$ there is a unique morphism $h$ such that $P(h) = 1_{\Pi'}$ making the following diagram to commute:




Note that, in general, restriction does not preserve reachable states. Often, thus, the result of a restriction is itself restricted to its reachable part.

**Relabelling.** In the same group of *interface-modifier* operators, is *relabelling*, which renames the labels of a PLTS according to a total function $\lambda : \Pi \to \Pi'$.

**Definition 9.** *Let $T = \langle W, i, R, \Pi \rangle$ be a PLTS, and $\lambda : \Pi' \to \Pi$ be a total function. The **relabelling** of $T$ according to $\lambda$, $T\{\lambda\}$ is the PLTS $\langle W, i, R', \Pi' \rangle$ where $R' = \{(w, \lambda(a), w', \alpha, \beta) \mid (w, a, w', \alpha, \beta) \in R\}$.*

Dually to the previous case, there is a morphism $f = (1_W, \lambda)$ from $T$ to $T\{\lambda\}$ which is the cocartesian lifting of $\lambda$ ($= P(f)$).

**Parallel composition.** The product of two PLTSs combines their state spaces and includes all *synchronous* transitions, triggered by the simultaneous occurrence of an action of each component, as well as *asynchronous* ones in which a transition in one component is paired with an *idle* transition, labelled by $\perp$, in the other. Formally,

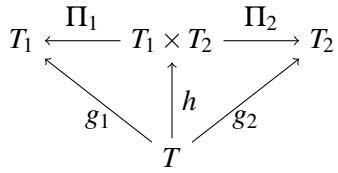
**Definition 10.** *Let $T_1 = \langle W_1, i_1, R_1, \Pi_1 \rangle$ and $T_2 = \langle W_2, i_2, R_2, \Pi_2 \rangle$ be two PLTS. Their **parallel composition** $T_1 \times T_2$ is the PLTS $\langle W_1 \times W_2, (i_1, i_2), R, \Pi' \rangle$, such that $\Pi' = \Pi_1 \times_\perp \Pi_2 = \{(a, \perp) \mid a \in \Pi_1\} \cup \{(\perp, b) \mid b \in \Pi_2\} \cup \{(a, b) \mid a \in \Pi_1, b \in \Pi_2\}$, and $(w, a, w', \alpha, \beta) \in R$ if and only if $(\pi_1(w), \pi_1(a), \pi_1(w'), \alpha_1, \beta_1) \in {R_1}^\perp$, $(\pi_2(w), \pi_2(a), \pi_2(w'), \alpha_2, \beta_2) \in {R_2}^\perp$, $\alpha = min(\alpha_1, \alpha_2)$ and $\beta = max(\beta_1, \beta_2)$.*

**Lemma 3.** *Parallel composition is the product construction in* $\mathsf{Pt}$.

*Proof.* In the diagram below let $g_i = (\sigma_i, \lambda_i)$, for $i = 1, 2$, and define $h$ as $h = (\langle \sigma_1, \sigma_2 \rangle, \langle \lambda_1, \lambda_2 \rangle)$, where $\langle f_1, f_2 \rangle(x) = (f_1(x), f_2(x))$ is the universal arrow in a product diagram in $\mathsf{Set}$. Clearly, $h$ lifts universality to $\mathsf{Pt}$, as the unique arrow making the diagram to commute. It remains show it is indeed an arrow in the category. Indeed, let $T = \langle W, i, R, \Pi \rangle$, $T_1 = \langle W_1, i_1, R_1, \Pi_1 \rangle$, and define $T_1 \times T_2 = \langle W_1 \times W_2, (i_1, i_2), R', \Pi' \rangle$ according to defintion 10. Thus, for each $(w, a, w', \alpha, \beta) \in R$, there is a transition $(\sigma_1(w), \lambda_1(a), \sigma_1(w'), \alpha_1, \beta_1) \in {R_1}^\perp$ such that $\alpha \leq \alpha_1$ and $\beta \geq \beta_1$; and also a transition $(\sigma_2(w), \lambda_2(a), \sigma_2(w'), \alpha_2, \beta_2) \in {R_2}^\perp$ such that $\alpha \leq \alpha_1$ and $\beta \geq \beta_2$. Moreover, there is a transition

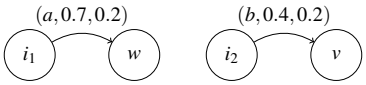
$$(\langle \sigma_1, \sigma_2 \rangle(w), \langle \lambda_1, \lambda_2 \rangle(a), \langle \sigma_1, \sigma_2 \rangle(w'), min(\alpha_1, \alpha_2), max(\beta_1, \beta_2)) \in R'$$

Thus, there is a transition $(\langle \sigma_1, \sigma_2 \rangle(w), \langle \lambda_1, \lambda_2 \rangle(a), \langle \sigma_1, \sigma_2 \rangle(w'), \alpha', \beta')) \in R'$, for any $(w, a, w', \alpha, \beta) \in R$ , such that $\alpha \leq \alpha'$ and $\beta \geq \beta'$. Furthermore, $\langle \sigma_1, \sigma_2 \rangle(i) = (\sigma_1(i), \sigma_2(i)) = (i_1, i_2)$. This establishes $h$ as a $\mathsf{Pt}$ morphism.

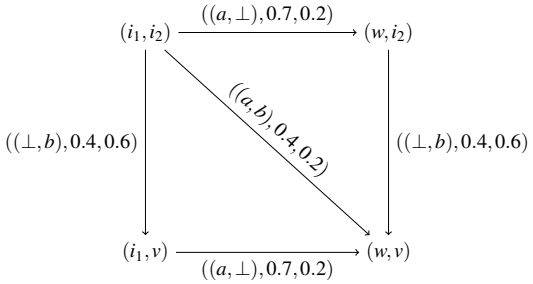



□

**Example 6.** *Consider the two PLTSs, $T_1$ and $T_2$, depicted below.*



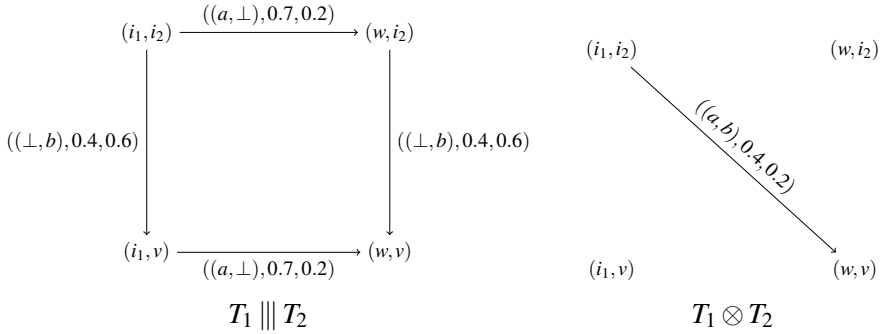

*Their product $T$ is the PLTS*




A suitable combination of parallel composition and restriction may enforce different synchronization disciplines. For example, *interleaving* or *asynchronous product* $T_1 \,|||\, T_2$ is defined as $(T_1 \times T_2) \upharpoonright \lambda$ with the inclusion $\lambda : \Pi \to \Pi_1 \times_\bot \Pi_2$ for $\Pi = \{(a, \bot) \mid a \in \Pi_1\} \cup \{(\bot, b) \mid b \in \Pi_2\}$. This results in a PLTS $\langle W_1 \times W_2, (i_1, i_2), R, \Pi \rangle$ such that $R = \{(w, a, w', \alpha, \beta) \in R' \mid a \in \Pi\}$.

Similarly, the *synchronous product* $T_1 \otimes T_2$ is also defined as $(T_1 \times T_2) \upharpoonright \lambda$, taking now $\Pi = \{(a, b) \mid a \in \Pi_1 \text{ and } b \in \Pi_2\}$ as the domain of $\lambda$.

**Example 7.** *Interleaving and synchronous product of $T_1$ and $T_2$ as in Example 8, are depicted below.*




**Sum.** The sum of two PLTSs corresponds to their non-determinisitic composition: the resulting PLTS behaves as either of its components. Formally,

**Definition 11.** *Let $T_1 = \langle W_1, i_1, R_1, \Pi_1 \rangle$ and $T_2 = \langle W_2, i_2, R_2, \Pi_2 \rangle$ be two PLTSs. Their sum $T_1 + T_2$ is the PLTS $\langle W, (i_1, i_2), R, \Pi_1 \cup \Pi_2 \rangle$, where*

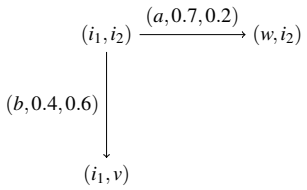
– $W = (W_1 \times \{i_2\}) \cup (\{i_1\} \times W_2)$ ,

– $t \in R$ *if and only if there exists a transition* $(w,a,w',\alpha,\beta) \in R_1$ *such that* $t = (\iota_1(w),a,\iota_1(w'),\alpha,\beta)$, *or a transition* $(w,a,w',\alpha,\beta) \in R_2$ *such that* $t = (\iota_2(w),a,\iota_2(w'),\alpha,\beta)$

*where* $\iota_1$ *and* $\iota_2$ *are the left and right injections associated to a coproduct in* Set*, respectively.*

Sum is actually a coproduct in Pt (the proof follows the argument used for the product case), making $T_1 + T_2$ dual to $T_1 \times T_2$.

**Example 8.** *The sum* $T_1 + T_2$*, for* $T_1, T_2$ *defined as in Example 8 is given by*




**Prefixing.** As a limited form of sequential composition, prefix appends to a pointed PLTS a new initial state and a new transition to the previous initial state, after which the system behaves as the original one.

**Definition 12.** *Let* $T = \langle W,i,R,\Pi\rangle$ *be a PLTS and* $w_{new}$ *a fresh state identifier not in* $W$*. Given an action* $a$*, and* $\alpha,\beta \in [0,1]$*, the prefix* $(a,\alpha,\beta)T$ *is defined as* $\langle W \cup \{w_{new}\}, w_{new}, R', \Pi \cup \{a\}\rangle$ *where* $R' = R \cup (w_{new},a,i,\alpha,\beta)$.

Since it is not required that the prefixing label is distinct from the ones in the original system, prefixing does not extend to a functor in Pt, as illustrated in the counterexample below. This is obviously the case for a category of classical labelled transition systems as well. In both cases, however, prefix extens to a functor if the corresponding categories are restricted to action-preserving morphisms, i.e. in which the action component of a morphism is always an inclusion

**Example 9.** *Consider two pointed PLTS* $T_1$ *and* $T_2$


*connected by a morphism* $(\sigma,\lambda) : T_1 \to T_2$ *such that* $\sigma(i_1) = i_2$*,* $\sigma(w) = v$ *and* $\lambda(a) = b$*. Now consider the prefixes* $(a,1,0)T_1$ *and* $(a,1,0)T_2$ *depicted below.*


*Clearly, a mapping from the actions in* $(a,1,0)T_1$ *to the actions in* $(a,1,0)T_1$ *does not exist so neither exists a morphism between the two systems.*

**Functorial extensions.** Other useful operations between PLTSs, typically acting on transitions' positive and negative weights, and often restricted to PLTSs over a specific residuated lattice, can be defined functorially in Pt. An example involving a PLTS defined over a Gödel algebra is an operation that uniformly increases or decreases the value of the positive (or the negative, or both) weight in all transitions. Let

$$a \oplus b = \begin{cases} 1 \text{ if } a+b \geq 1 \\ 0 \text{ if } a+b \leq 0 \\ a+b \text{ otherwise} \end{cases}$$

Thus,

**Definition 13.** *Let $T = \langle W, i, R, \Pi \rangle$ be a PLTS. Taking $v \in [-1,1]$, the **positive $v$-approximation** $T_{\oplus v}^{+}$ is a PLTS $\langle W, i, R', \Pi \rangle$ where*

$$R' = \{(w, \pi, w', \alpha \oplus v, \beta) \mid (w, \pi, w', \alpha, \beta) \in R\}.$$

The definition extends to a functor in Pt which is the identity in morphisms. Similar operations can be defined to act on the negative accessibility relation or both.

Another useful operation removes all transitions in a pointed PLTS for which the positive accessibility relation is below a certain value and the negative accessibility relation is above a certain value. Formally,

**Definition 14.** *Let $T = \langle W, i, R, \Pi \rangle$ be a pointed PLTS, and $p, n \in [0,1]$. The **purged** PLTS $T_{p\uparrow\downarrow n}$ is defined as $\langle W, i, R', \Pi \rangle$ where*

$$R' = \{(w, \pi, w', \alpha, \beta) \mid (w, \pi, w', \alpha, \beta) \in R \text{ and } \alpha \geq p \text{ and } \beta \leq m\}$$

Clearly, the operation extends to a functor in Pt, mapping morphisms to themselves.

## 5 An application to quantum circuit optimization

In a quantum circuit [10] decoherence consists in decay of a qubit in superposition to its ground state and may be caused by distinct physical phenomena. A quantum circuit is effective only if gate operations and measurements are performed to superposition states within a limited period of time after their preparation. In this section pointed PLTS will be used to model circuits incorporating qubit decoherence as an error factor. Typically, coherence is specified as an interval corresponding to a worst and a best case. We employ the two accessibility relations in a PLTS to model both scenarios simultaneously.

An important observation for the conversion of quantum circuits to PLTS is that quantum circuits always have a sequential execution. Simultaneous operations performed to distinct qubits are combined using the tensor product $\otimes$ into a single operation to the whole collection of qubits which forms the state of the circuit. The latter is described by a sequence of executions $e_1, e_2, e_3, \ldots$ where each $e_i$ is the tensor product of the operations performed upon the state at each step. The conversion to a PLTS is straightforward, labelling each transition by the tensor of the relevant gates $O_1 \otimes \cdots \otimes O_m$, for $m$ gates involved, but for the computation of the positive and negative accessibility relations, $r^+$ and $r^-$.

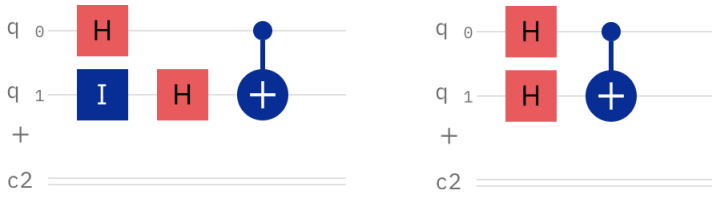
The weights of a transition corresponding to the application of a gate $O$ acting over $n$ qubits $q_1$ to $q_n$ are given by

$$v(O) = \begin{cases} (1,0) \text{ if qubits } q_1, \cdots q_n \text{ are in a definite state} \\ (\text{Max}_i\, f_{\max}(q_i), \text{Min}_i\, f_{\min}(q_i)) \text{ otherwise} \end{cases}$$

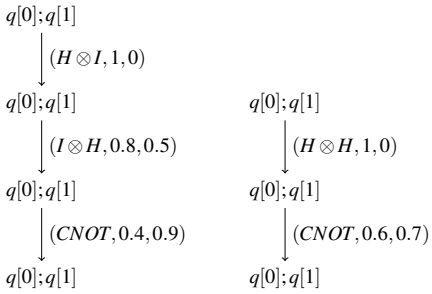
where $f_{\max}(q) = \frac{\tau_{\max}(q) - \tau_{\text{prep}}(q)}{100}$ and $f_{\min}(q) = \frac{\tau_{\min}(q) - \tau_{\text{prep}}(q)}{100}$, $\tau_{\max}(q)$ and $\tau_{\min}(q)$ are the longest and shortest coherence times of $q$, respectively, and $\tau_{\text{prep}}(q)$ is the time from the preparation of $q$'s superposition to the point after the execution of $O$. The latter are fixed for each type of quantum gate; reference [14] gives experimentally computed values for them as well as for maximum and minimum values for qubit decoherence.

Consider, now, a transition $t$ labelled by a $O_1 \otimes \ldots \otimes O_m$ Then, $r^+ = \text{Max}_{i=1}^{n}\{\pi_1(v(O_i))\}$ and $r^- = 1 - \text{Min}_{i=1}^{n}\{\pi_2(v(O_i))\}$.

**Example 10.** *Consider the following circuits designed with IBM Quantum Composer:*




*Assume that the execution time of a single qubit gate is* $\tau_G = 20\mu s$ *and of a two qubit gate is* $2\tau_G = 40\mu s$ *[14], and that both qubits have the same coherence times* $\tau_{max}(q_1) = \tau_{max}(q_2) = 100\mu s$ *and* $\tau_{min}(q_1) = \tau_{min}(q_2) = 70\mu s$*. Thus the circuit on the left (resp. right) translates into* $T_1$ *(on the left) and* $T_2$ *(on the right).*




*As both circuits implement the same quantum algorithm and our focus is only on the effectiveness of the circuits, we may abstract from the actual sequences of labels and consider instead* $T_1\{\lambda\}$ *and* $T_2\{\lambda\}$*, for* $\lambda$ *mapping each label to a unique label* $\star$*. Their maximal weighted traces* [2] *are*

$$t_{T_1\{\lambda\}} = \langle [*,*,*], 0.4, 0.9 \rangle \text{ and } t_{T_2\{\lambda\}} = \langle [*,*,*], 0.6, 0.7 \rangle$$

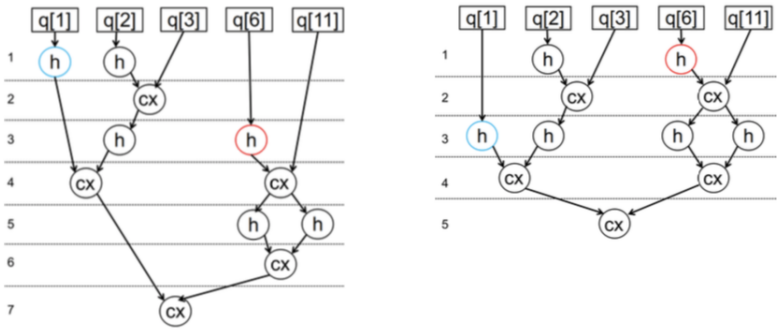
*Clearly* $t_{T_1\{\lambda\}}$ *is a weighted subtrace of* $t_{T_2\{\lambda\}}$*, therefore suggesting a criteria for comparing the effectiveness of circuits. Indeed, a circuit is more effective (i.e. less affected by qubit decoherence) than other if the maximal weighted trace of its (relabelled) PLTS is a weighted subtrace of the corresponding construction in the other.*

*The second circuit is obviously more efficient than the first. This suggests we could use the weighted subtrace relation as a metric to compare circuit quality, for circuits implementing equivalent algorithms.*

Reference [14] introduces a tool which tried to transform a circuit so that the lifetime of quantum superpositions is shortened. They give several examples of circuits and show how the application of the tool results in a circuit performing the same algorithm but with a reduced error rate. Our next example builds on one of their examples, computes the corresponding PLTS and compare the maximal weighted traces.

**Example 11.** *Consider the following circuits reproduced from [14], which in ideal quantum devices would be indistinguishable.*

[2] Such maximal traces are easily identifiable given the peculiar shape of a PLTS corresponding to a quantum circuit.

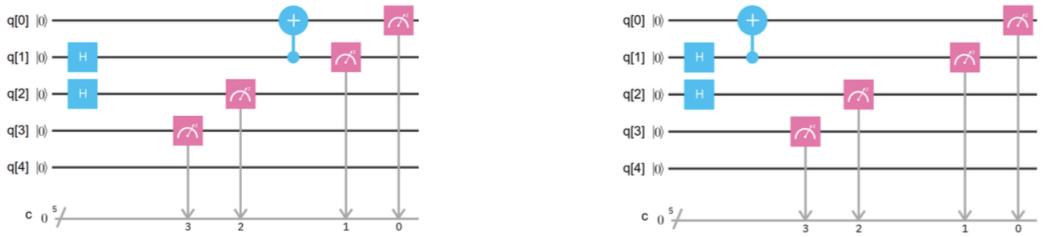



*These circuits are represented as*

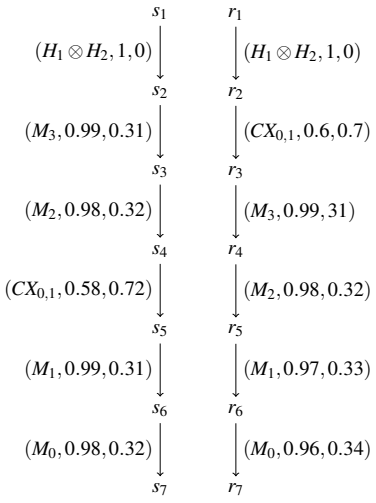
$$
\begin{array}{lll}
s_1 & & \\
\downarrow (H_1 \otimes H_2, 1, 0) & & \\
s_2 & & \\
\downarrow (CX_{2,3}, 0.6, 0.7) & & \\
s_3 & r_1 & \\
\downarrow (H_2 \otimes H_6, 0.8, 0.5) & \downarrow (H_2 \otimes H_6, 1, 0) & \\
s_4 & r_2 & \\
\downarrow (CX_{1,2} \otimes CX_{6,11}, 0, 1) & \downarrow (CX_{2,3} \otimes CX_{6,11}, 0.6, 0.7) & \\
s_5 & r_3 & \\
\downarrow (H_6 \otimes H_1, 0.8, 0.5) & \downarrow (H_1 \otimes H_2 \otimes H_6 \otimes H_1 1, 0.8, 0.5) & \\
s_6 & r_4 & \\
\downarrow (CX_{6,11}, 0.6, 0.7) & \downarrow (CX_{1,2} \otimes CX_{6,11}, 0.6, 0.7) & \\
s_7 & r_5 & \\
\downarrow (CX_{2,6}, 0, 1) & \downarrow (CX_{2,6}, 0.6, 0.7) & \\
s_8 & r_6 &
\end{array}
$$

*where H and CX are indexed by the numeric identifiers of the qubit(s) to which they apply in each execution step. The maximal weighted trace of the (relabelled PLTS corresponding to) circuit in the right, $\langle [*,*,*,*,*,*,*], 0.6, 0.7 \rangle$, is a weighted subtrace of the one corresponding to circuit in the left, $\langle [*,*,*,*,*], 0, 1 \rangle$. Thus, the former circuit is more effective than the latter, as experimentally verified in [14].*

**Example 12.** *As a final example consider two circuits differing only on the time points in which measurements are placed.*



*The corresponding PLTS, computed again with the values given in reference (where execution time of a measurement is $\tau_M = 300ns \sim 1\mu s$), are depicted below*




*The maximal weighted trace $\langle[*,*,*,*,*,*,],0.6,0.7\rangle$ corresponding to the circuit on the right is a weighted subtrace of the corresponding one for the circuit on the left, $\langle[*,*,*,*,*,*,],0.58,0.72\rangle$. This shows that measuring can be safely postponed to the end of a circuit, as experimentally verified.*

## 6 Conclusions and future work

The paper introduced a category of a new kind of labelled transition systems able to capture both *vagueness* and *inconsistency* in software modelling scenarios. The structure of this category was explored to define a number of useful operators to build such systems in a compositional way. Finally, PLTS were used to model effectiveness concerns in the analysis of quantum circuits. In this case the weight corresponding to the 'presence' of a transition captures an index measuring its effectiveness assuming the best case value for qubit decoherence. On the other hand, the weight corresponding to the 'absence' of a transition measures the possibility of non-occurrence, assuming qubit decoherence worst case value.

A lot remains to be done. First of all, a process logic, as classically associated to labelled transition systems [12], i.e. a modal logic with label-indexed modalities, can be designed for pointed PTLS. This will provide not only yet another behavioural equivalence, based on the set of formulas satisfied by two systems, but also a formal way to express safety and liveness properties of these systems.

This will be extremely useful to express and verify properties related to the effectiveness of quantum circuits, therefore pushing further the application scenario proposed in section 5. Finally, automating the construction of a pointed PLTS for a given circuit, parametric on the different qubit coherence and gate execution time found experimentally, and adding a prover for the logic suggested above, will provide an interesting basis to support quantum circuit optimization. Reliable, mathematically sound approaches and tools to support quantum computer programming and verification will be part of the quantum research agenda for the years to come. Indeed, their lack may put at risk the expected quantum advantage of the new hardware.

# References


[1] Agudelo, J.C.A., Carnielli, W.A.: Paraconsistent machines and their relation to quantum computing. J. Log. Comput. **20**(2), 573–595 (2010)

[2] Akama, S. (ed.): Towards Paraconsistent Engineering, Intelligent Systems Reference Library, vol. 110. Springer (2016)

[3] Baeten, J.C.M., Basten, T., Reniers, M.A.: Process Algebra: Equational theories of communicating processes. Cambridge Tracts in Theoretical Computer Science (50), Cambridge University Press (2010)

[4] Chiara, M.L.D., Giuntini, R.: Paraconsistent ideas in quantum logic. Synth. **125**(1-2), 55–68 (2000)

[5] da Costa, N.C.A., Krause, D., Bueno, O.: Paraconsistent logics and paraconsistency. In: Jacquette, D. (ed.) Handbook of the Philosophy of Science (Philosophy of Logic). pp. 791–911. Elsevier (2007)

[6] da Costa, N.C.A., Krause, D.: Physics, inconsistency, and quasi-truth. Synth. **191**(13), 3041–3055 (2014)

[7] Cruz, A., Madeira, A., Barbosa, L.S.: A logic for paraconsistent transition systems. In: Indrzejczak, A., Zawidzki, M. (eds.) Proceedings of the 10th International Conference on Non-Classical Logics. Theory and Applications, NCL 2022, Łódź, Poland, 14-18 March 2022. EPTCS, vol. 358, pp. 270–284 (2022). https://doi.org/10.4204/EPTCS.358.20, `https://doi.org/10.4204/EPTCS.358.20`

[8] Cruz, A.L.R.: Exploring paraconsistent logics for quantum programs. MSc Thesis in Engineering Physics, DI, Universidade do Minho (2021)

[9] Jaśkowski, S.: Propositional calculus for contradictory deductive systems. Studia Logica **24**(1), 143–157 (1969)

[10] Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information (10th Anniversary Edition). Cambridge University Press (2010)

[11] Preskill, J.: Quantum computing in the nisq era and beyond. Quantum **2**(79), 87–95 (2018)

[12] Stirling, C.: Modal and Temporal Properties of Processes. Texts in Computer Science, Springer (2001)

[13] Winskel, G., Nielsen, M.: Models for concurrency. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science (vol. 4): Semantic Modelling, pp. 1–148. Oxford Science Publications (1995)

[14] Zhang, Y., Deng, H., Li, Q., Song, H., Nie, L.: Optimizing quantum programs against decoherence: Delaying qubits into quantum superposition. In: 2019 Int. Symp. Theoretical Aspects of Software Engineering (TASE). IEEE (Jul 2019)

# *ReLo*: a dynamic logic to reason about Reo circuits*

Erick Grilo
Instituto de Computação
Universidade Federal Fluminense
simas_grilo@id.uff.br

Bruno Lopes
Instituto de Computação
Universidade Federal Fluminense
bruno@ic.uff.br

Critical systems require high reliability and are present in many domains. They are systems in which failure may result in financial damage or even loss of lives. Standard techniques of software engineering are not enough to ensure the absence of unacceptable failures and/or that critical requirements are fulfilled. Reo is a component-based modelling language that aims to provide a framework to build software based on existing pieces of software, which has been used in a wide variety of domains. Its formal semantics provides grounds to certify that systems based on Reo models satisfy specific requirements (i.e., absence of deadlocks). Current logical approaches for reasoning over Reo require the conversion of formal semantics into a logical framework. *ReLo* is a dynamic logic that naturally subsumes Reo's semantics. It provides a means to reason over Reo circuits. This work extends *ReLo* by introducing the iteration operator, and soundness and completeness proofs for its axiomatization.The core aspects of this logic are also formalized in the Coq proof assistant.

## 1 Introduction

In software development, service-oriented computing [31] and model-driven development [6] are examples of techniques that take advantage of software models. The first technique advocates computing based on preexisting systems (services) as described by Service-Oriented Architecture (SOA), while the latter is a development technique that considers the implementation of a system based on a model. A model is an abstraction of a system (or some particular portion of it) in a specific language, which will be used as a specification basis for the system's implementation. It can be specified in languages such as Unified Modeling Language (UML) or formal specification languages like B [1] or Alloy [16]. Researchers also have applied approaches such as formal methods in software development to formalize and assure that certain (critical) systems have some required properties [19, 30].

Reo [2] is a prominent modelling language, enabling coordination of communication between interconnected systems without focusing on their internal properties. Reo models are compositionally built from base connectors, where each connector in Reo stands for a specific communication pattern. Reo has proven to be successful in modeling the organization of concurrent systems' interaction, being used in a variety of applications, from process modeling to Web-Services integration [4] and even in the construction of frameworks to verify specifications in Reo [22, 34].

Reo's ability to model communication between software interfaces has also attracted research on verification of Reo circuits, resulting in many different formal semantics [17] like automata-based models [3, 7, 23], coalgebraic models [2], Intuitionistic Logic with Petri Nets [12] (to name a few), and some of their implementations [22, 33, 35, 26, 29, 36, 23]. However, as far as the authors are concerned, there is no logic apart from *ReLo* [13] to specific reason about Reo models naturally, where the usage of other logic-based approaches requires conversion between different formal semantics.

*This work was supported by CNPq and FAPERJ.

This work extends *ReLo* [13] by introducing an iteration operator and the soundness and completeness proofs of its axiomatic system. A prototypical implementation of this framework in Coq proof assistant, enabling the verification of properties of Reo programs in *ReLo* within a computerized environment is available at http://github.com/frame-lab/ReoLogicCoq.

This work is structured as follows. Section 3 discusses briefly a related logic formalism with the one hereby proposed and introduces Reo modelling language, along with some examples. Section 4 discuss *ReLo*'s main aspects, from its core definitions (such as language, models, transitions firing) and its soundness and completeness proofs. Finally, Section 5 closes the work by discussing the obtained results and assessing possible future work.

## 2 Related work

The fact that Reo can be used to model many real-world situations has attracted attention from researchers all around the world, resulting in a great effort directed in formalizing means to verify properties of Reo models [18, 32, 20, 22, 28, 27, 17]. Such effort also resulted in the proposal of several formal semantics for this modelling language [17], varying from operational semantics to coalgebric models.

One of the most known formal semantics for Reo consists of Constraint Automata [8], an operational semantic in which Reo connectors are modelled as automata for $TDS$-languages [5]. It enables reasoning over the data flow of Reo connectors and when they happened. Constraint Automata have been extended to some variants which aim to enrich the reasoning process by capturing properties like the timing of the data flows or possible actions over the data, respectively as Timed Constraint Automata [22] and Action Constraint Automata [21]. Some of them are briefly discussed below, along with other formal semantics for Reo.

The approach presented by Klein et al. [18] provides a platform to reason about Reo models using Vereofy,[1] a model checker for component-based systems, while Pourvatan et al. [32] propose an approach to reason about Reo models employing symbolic execution of Constraint Automata. Kokash & Arbab [20] formally verify Long-Running Transactions (LRTs) modelled as Reo connectors using Vereofy, enabling expressing properties of these connectors in logics such as Linear Temporal Logic (LTL) or a variant of Computation Tree Logic (CTL) named Alternating-time Stream Logic (ASL). Kokash et al. [22] use mCRL2 to encode Reo's semantics in Constraint Automata and other automata-based semantics, encoding their behaviour as mCRL2 processes and enabling the expression of properties regarding deadlocks and data constraints which depend upon time. mCRL2 also supports model-checking of Reo in a dynamic logic (with fixed points), where modalities are regular expressions, atomic actions are sets of nodes that fire at the same time. Mouzavi et al. [28] propose an approach based on Maude to model checking Reo models, encoding Reo's operational semantics of the connectors.

Proof assistants have been used to reason about Reo connectors [25, 26, 29, 35, 36, 14]. The approaches adopted by Li et al. [25, 35, 14] are among the ones that employ Coq to verify Reo models formally. In [25] a formalization of four of the Reo canonical connectors (Sync, FIFO1, SyncDrain, and LossySync) along with an LTL-based language defined as an inductive type in Coq is presented, while [35] proposes the formalization of five Reo canonical channels and a procedure that creates composite channels by logical conjunction of the connectors modelled.

In [14], a framework to provide means of graphically model Reo connectors and validate the generated model in Constraint Automata using Coq and NuSMV[2] is discussed. It also enables the automatic

[1] http://www.vereofy.de
[2] https://nusmv.fbk.eu/

generation of Coq code to a Haskell model employing the Coq code extraction apparatus. When restricting the works considering logics and Reo, as far as the authors know there is only the work by [12] which focuses on formalizing the semantics of Reo connectors Sync, LossySync, FIFO1, SyncDrain, AsyncDrain, Filter, Transform, Merger, and Replicator in terms of zero-safe Petri nets [11], a special class of Petri-nets with two types of places: zero and stable places. This encoding is then converted to terms in Intuitionistic Temporal Linear Logic, enabling reasoning about Reo connectors in this logic.

# 3 Background

This section provides a succinct overview of Reo [2, 3], considering its main characteristics and a modelling examples as it is the target language *ReLo* provides a formal semantic to reason over.

## 3.1 The Reo modelling language

As a coordination model, Reo focuses on connectors, their composition, and how they behave, not focusing on particular details regarding the entities that are connected, communicate, and interact through those connectors. Connected entities may be modules of sequential code, objects, agents, processes, web services, and any other software component where its integration with other software can be used to build a system [2]. Such entities are defined as component instances in Reo.

Channels in Reo are defined as a point-to-point link between two distinct nodes, where each channel has its unique predefined behavior. Each channel in Reo has exactly two ends, which can be of the following types: the source end, which accepts data into the channel, and the sink end, which dispenses data out of the channel. Channels are used to compose more complex connectors, being possible to combine user-defined channels amongst themselves and with the canonical connectors provided by Baier et al. [8]. Figure 1 shows the basic set of connectors as presented by Kokash et al. [22].

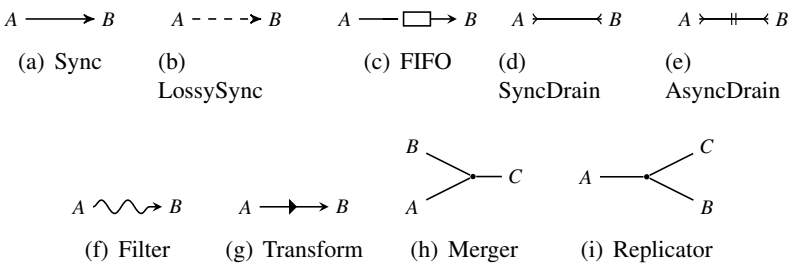



Figure 1: Canonical Reo connectors

Channel ends can be used by any entity to send/receive data, given that the entity belongs to an instance that knows these ends. Entities may use channels only if the instance they belong to is connected to one of the channel ends, enabling either sending or receiving data (depending on the kind of channel end the entity has access to).

The bound between a software instance and a channel end is a logical connection that does not rely on properties such as the location of the involved entities. Channels in Reo have the sole objective to enable the data exchange following the behaviour of the connectors composing the channel, utilizing I/O operations predefined for each entity in an instance. A channel can be known by zero or more instances at a time, but its ends can be used by at most one entity at the same time.

Figure 2 introduces a Reo connector known as Sequencer[3]. It models the data flow between three entities sequentially. The data flows from the first FIFO connector (a buffer), which will be sequentially synchronized with entities in port names names A, B, and C. The Sequencer can be used to model scenarios where processes sequentially interact between themselves.

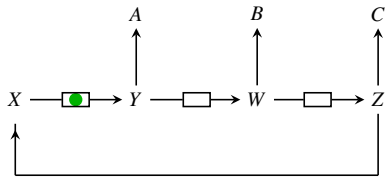



Figure 2: Modelling of the Sequencer in Reo

In short, Reo circuits may be understood as data flowing from different interfaces (i.e., port names connected to a node), where the connector itself models the communication pattern between two of these interfaces. A *ReLo* program is composed of one or more Reo connectors as introduced in Figure 1.

## 4 A *ReLo* Primer

*ReLo* [13] was tailored to subsume Reo models' behaviour naturally in a logic, without needing any mechanism to convert a Reo model denoted by one of its formal semantics to some logical framework. Each basic Reo connector is modelled in the logic's language, which is defined as follows.

**Definition 1** (*ReLo*'s language)**.** The language of *ReLo* consists of the following:

- An enumerable set of propositions $\Phi$.
- Reo channels as denoted by Figure 1
- A set of port names $\mathcal{N}$
- A sequence $Seq_\Pi = \{\varepsilon, s_1, s_2, \dots\}$ of data flows in ports of a *ReLo* program $\Pi$ (defined below). We define $s_i \leq s_j$ if $s_i$ is a proper (i.e., $s_j$ contains all of $s_i$'s data). Each sequence $s_i$ denotes the data flow of the Reo program $\Pi$ (i.e., all ports that have data synchronized at a specific moment in time) and $\varepsilon$ is the empty sequence
- Program composition symbol : $\odot$
- A sequence $t$ of data flows of ports $p$ with data values $\{0,1\}$, which denotes whether $p$ contains a data item. This describes a data flow occurring in the Reo channel. A BNF describing $t$ is defined as follows:
  $\langle t \rangle$ ::= $\langle portName \rangle \langle data \rangle , \langle t \rangle \mid \langle data \rangle \langle portName \rangle \langle data \rangle , \langle t \rangle$
  $\mid \langle data \rangle \langle portName \rangle \langle data \rangle \mid \langle portName \rangle \langle data \rangle$
  $\langle portName \rangle$ ::= $p \in \mathcal{N}$
  $\langle data \rangle$ ::= $0 \mid 1$
- Iteration operator $^\star$

A *ReLo* program is defined as any Reo model built from the composition of Reo channels $\pi_i$. In *ReLo* their composition is $\Pi = (f, b)$, $\Pi = \pi_1 \odot \pi_2 \odot \cdots \odot \pi_n$, and $\pi_i = (f_1, b_1)$. $\odot$ follows the same notion of Reo composition, by "gluing" sink nodes of a connector to the source nodes of the other connector.

[3] http://arcatools.org/reo

The set $f$ is the set of connectors $p$ of the model where data flows in and out of the channel (the connector has at least a source node and a sink node), namely Sync, LossySync, FIFO, Filter, Transform, Merger and Replicator. The set $b$ is the set of blocking channels (channels without sink nodes whose inability to fire prevents the remainder of connectors related to their port names from fire), namely SyncDrain and AsyncDrain.

The following is a simple yet intuitive example of the structure of data flows in *ReLo*. Let the sequence $t$ be $t = \{A1, B1C\}$. It states that the port $A$ has the data item 1 in its current data flow, while there is a data item 1 in the FIFO between $B$ and $C$.

**Definition 2** (*ReLo* formulae)**.**
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 $\top \equiv \neg\bot, \phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi), \phi \rightarrow \psi \equiv \neg\phi \vee \psi$ and $[t, \pi]\varphi \equiv \neg\langle t, \pi \rangle \neg\phi$, where $\pi$ is some Reo program and $t$ a data flow.

The connectors in Figure 3 exemplify compound Reo connectors. The model SyncFIFO is composed of a FIFO and a Sync connector in which the data leaving the FIFO is sent to $C$ from $B$ synchronously. Suppose that there is data in the FIFO and in port $B$ ($t = \{A1B, B0\}$). If the FIFO from $A$ to $B$ is processed first then the Sync between $B$ and $C$, the data flow in $B$ will be overwritten before it is sent to $C$, which is not the correct behaviour. The Sync from $B$ to $C$ must fire before the FIFO from $A$ to $B$.

Another example is denoted by the model Sync2Drain. Suppose there is data only in port name $A$ ($t = \{A1\}$). If the Sync from $B$ to $A$ is evaluated first then the SyncDrain between $B$ and $C$, the restriction imposed by the fact that the condition required for the SyncDrain to fire was not met (as $C$'s data flow differs from $B$'s at this moment) is not considered, and data will wrongly flow from $B$ to $A$. The SyncDrain must be first evaluated before all flows as they may block the flow from data of its ports to other channels.


(a) SyncFIFO (b) Sync2Drain

Figure 3: Examples of Reo models

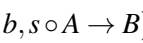
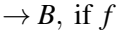
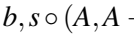
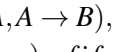
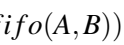
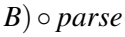
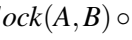
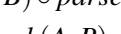
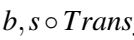
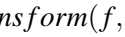
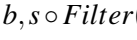
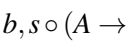
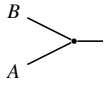
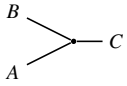
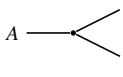
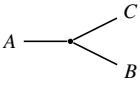
The next definition maps each canonical connector that composes a Reo model to a *ReLo* program. The left hand side of each mapping rule in Definition 3 is the atomic Reo connector, while the right hand size is the resulting *ReLo* atomic program $\pi_i = (f_i, b_i)$, with the same behaviour as of the Reo connector.

**Definition 3** (*parse* base cases)**.** Each canonical Reo connector is mapped to a *ReLo* program in *parse*:

-  to $A \rightarrow B$
-  to $(A, A \rightarrow B)$
-  to $fifo(A,B)$
-  to $SBlock(A,B)$
-  to $ABlock(A,B)$
-  to $Transform(f,A,B)$, $f$: $Data \rightarrow Data$ is a transformation function.
-  to $Filter(P,A,B)$, $P$ is a logical predicate over the data item in $A$.
-  to $(A \rightarrow C, B \rightarrow C)$
-  to $(A \rightarrow B, A \rightarrow C)$

Considering that each *ReLo* program $\Pi$ is the composition of programs $\pi_1 \odot \pi_2, \odot \cdots \odot \pi_n, \pi_i = (f_i, b_i)$ as Reo programs, *parse* is formalized in Definition 4. The symbol $\circ$ denote the addition of an element to $s$, the resulting set of *parse*'s processing.

**Definition 4** (*parse* function)**.** The function that interprets the execution of a *ReLo* program is defined as $parse(f,b,s)$. We define $\varepsilon$ as an abbreviation to denote when there is no *ReLo* program left to process (i.e. the base case when no program is parametrized). Its outcome is detailed as below.

- $s$, if $f = b = \varepsilon$
- $parse(f_j,b,s \circ A \to B)$, if $f =$  $\odot f_j$
  - $s \circ A \to B$, if $f =$ 
- $parse(f_j,b,s \circ (A,A \to B))$, if $f =$  $\odot f_j$
  - $s \circ (A,A \to B)$, if $f =$ 
- $parse(f_j,b,s) \circ fifo(A,B)$, if $f =$  $\odot f_j$
  - $(s \circ fifo(A,B))$, if $f =$ 
- $SBlock(A,B) \circ parse(f,b_j,s)$, if $b =$  $\odot b_j$
  - $(SBlock(A,B) \circ s)$, if $b =$ 
- $ABlock(A,B) \circ parse(f,b_j,s)$, if $b =$  $\odot b_j$
  - $(ABlock(A,B) \circ s)$, if $b =$ 
- $parse(f_j,b,s \circ Transform(f,A,B))$, if $f =$  $\odot f_j$
  - $(Transform(f,A,B) \circ s)$, if $f =$ 
- $parse(f_j,b,s \circ Filter(P,A,B))$, if $f =$  $\odot f_j$
  - $(Filter(P,A,B) \circ s)$, if $f =$ 
- $parse(f_j,b,s \circ (A \to C,B \to C))$, if $f =$  $\odot f_j$
  - $(s \circ (A \to C,B \to C))$, if $f =$ 
- $parse(f_j,b,s \circ (A \to B,A \to C))$, if $f =$  $\odot f_j$
  - $(s \circ (a \to b,a \to c))$, if $f =$ 

We employ *parse* to interpret Reo programs $\Pi$ as a sequence of occurrences of possible data flow (where each flow corresponds to the execution of a Reo connector). These data flow may denote data transfer (*ReLo* programs $(A \to B)$ and $(A,A \to B)$, flow "blocks" induced by connectors such as SyncDrain and aSyncDrain (*ReLo* programs $SBlock(A,B)$ and $ABlock(A,B)$ — the first one requires that data flow synchronously through its ports, while the latter requires that data flow asynchronously through its ports). There is also the notion of a buffer introduced by FIFO connectors (*ReLo* program $fifo(A,B)$), which data flow into a buffer before flowing out of the channel, and merging/replicating data flow between ports, respectively denoted by channels Merger and Replicator (*ReLo* programs $(A \to C,B \to C)$ and $(A \to B,A \to C)$ respectively).

There are also special data flows, denoting the "transformation" of some data flowing from A to B as $Transform(f,A,B)$ which will apply $f$ with the data in $A$ before it sends $f(D_A)$ ($D_A$ denoting the data

item in A) to $B$, and the filtering of data flow by some predicate as $Filter(P,A,B)$, $P$ as a quantifiable-free predicate over the data item seen in $A$. Therefore, data will flow to $B$ only if $P(D_A)$ is satisfied.

After processing $\pi$ with *parse*, the interpretation of the execution of $\pi$ is given by $go(t,s,acc), go: s \times s \to s$, where $s$ is a string denoting the processed program $\pi$ as the one returned by *parse*, and $t$ is the initial data flow of ports of the Reo program $\pi$. The parameter *acc* holds all connectors of the Reo circuit that satisfy their respective required conditions for data to flow. In what follows we define $ax \prec t$ as an operator which states that $ax$ is in $t$, $ax$ a single data of a port and $t$ a structure containing data flows for ports $p \in \mathscr{N}$.

Example 1 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 then the data item in Y, the latter will be overwritten and the information is lost.

**Example 1.** let $\pi$ be the Reo program corresponding to the circuit in Fig. 2:

$\pi =$ X —□→ Y ⊙ Y ⟶ A ⊙ Y —□→ W ⊙ W ⟶ B ⊙ W —□→ Z ⊙ Z ⟶ C ⊙ Z ⟶ X

$parse(\pi,\{\}) = \{Y \to A; W \to B; Z \to C; Z \to X; fifo(X,Y); fifo(Y,W); fifo(W,Z)\}$

The usage of *parse* is required to eliminate problems regarding the execution order of $\pi$'s Reo channels, which could be caused by processing $\pi$ the way it is inputted (i.e., its connectors can be in any order). Consider, for example, the behavior of SyncDrain and aSyncDrain programs as "blocking" programs as discussed earlier. In a single step, they must be evaluated before the flow programs, because if they fail to execute due to missing requirements, data should not flow from their port names to other connectors. In a nutshell, *parse* organizes the program so this verification can be performed.

Therefore, the interpretation of a $\pi$ program processed by *parse* is performed by $go(t,s,acc)$, where $s$ is a string containing $\pi$ as processed by *parse*, $t$ is $\pi$'s initial data flow, and *acc* filters the connectors of the *ReLo* program that can be fired if the requirements to do so are met.

Definition 5 will check for each of the Reo connectors processed by *parse* satisfies the required condition to fire, following the connectors' behaviour. Operator $\prec$ denotes whether the data flow is within the current data flow $t$ being evaluated. It is also used to denote whether the program currently being evaluated in $s$ repeats in $\Pi$. Operator $\setminus$ denotes the removal of an connector from the accumulator *acc*.

**Definition 5** (Relation *go* for a single execution step)**.** We define $go(t,s,acc)$ as follows:

- $s = \varepsilon : fire(t,acc)$
- $s = A \to B \circ s'$ :
  - $go(t,s',acc \circ (A \to B))$, 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 $\begin{cases} Ax \prec t, \\ (A \to B) \not\prec s' \\ \exists s'_j \in acc \mid sink(s'_j) = B \end{cases}$
  - $go(t,s',acc)$, otherwise
- $s = (A, A \to B) \circ s'$ :
  - $go(t,s',acc \circ (A \to B)) \cup go(t,s',acc \circ (A \to A))$, 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 $\begin{cases} Ax \prec t, \\ (A \to B) \not\prec s' \\ \exists s'_j \in acc \mid sink(s'_j) = B \end{cases}$
- $go(t, s', acc)$, otherwise

- $s = fifo(A,B) \circ s'$ :
  - $go(t, s', acc \circ (AxB))$, iff $Ax \prec t, fifo(A,B) \not\prec s', (AxB) \not\prec acc$
  - $go(t, s', acc \circ (AxB \to Bx))$, iff $AxB \prec t, fifo(A,B) \not\prec s'$
  - $go(t, s', (acc \circ (AxB \to Bx)) \setminus s'_j) \cup go(t, s', acc)$, iff $\begin{cases} AxB \prec t, \\ fifo(A,B) \not\prec s', \\ \exists s'_j \in acc \mid sink(s'_j) = B \end{cases}$
  - $go(t, s', acc)$, otherwise
- $s = Sblock(A,B) \circ s'$ :
  - $go(t, s', acc)$, iff $\begin{cases} (Ax \prec t \wedge Bx \prec t) \vee (Ax \not\prec t \wedge Bx \not\prec t) \\ Sblock(A,B) \not\prec s' \end{cases}$
  - $go(t, halt(A,B,s'), acc)$, iff $\begin{cases} (Ax \prec t \wedge Bx \not\prec t) \vee (Ax \not\prec t \wedge Bx \prec t) \\ Sblock(A,B) \not\prec s' \end{cases}$
- $s = Ablock(A,b) \circ s'$ :
  - $go(t, s', acc)$, iff $\begin{cases} (Ax \not\prec t \wedge Bx \prec t) \vee (Ax \prec t \wedge Bx \not\prec t) \vee \\ (Ax \not\prec t \wedge Bx \not\prec t), Ablock(A,B) \not\prec s' \end{cases}$
  - $go(t, halt(A,B,s'), acc)$, iff $\begin{cases} (Ax \prec t \wedge Bx \prec t), \\ Ablock(A,B) \not\prec s' \end{cases}$
- $s = Transform(f,A,B) \circ s'$ :
  - $go(t, s', acc \circ (f(D_A) \to B))$, iff $\begin{cases} ax \prec t \\ Transform(f,A,B) \not\prec s' \end{cases}$
  - $go(t, s', (acc \circ (f(D_A) \to B)) \setminus s'_j) \cup go(t, s', acc)$, iff $\begin{cases} Ax \prec t, \\ Transform(f,A,B) \not\prec s' \\ \exists s'_j \in acc \mid sink(s'_j) = B \end{cases}$
  - $go(t, s', acc)$, otherwise
- $s = Filter(f,A,B) \circ s'$ :
  - $go(t, s', acc \circ (A \to B))$, iff $\begin{cases} Ax \prec t \\ P(D_A) \text{ holds} \\ Filter(f,A,B) \not\prec s' \end{cases}$
  - $go(t, s', (acc \circ (A \to B)) \setminus s'_j) \cup go(t, s', acc)$, iff $\begin{cases} Ax \prec t, \\ P(D_A) \text{ holds} \\ Filter(f,A,B) \not\prec s' \\ \exists s'_j \in acc \mid sink(s'_j) = B \end{cases}$
  - $go(t, s', acc)$, otherwise

The existing condition after each return condition of *go* denotes the case where two or more Reo connectors within a circuit have the same sink node. This implies that if both of their respective source

nodes have data flowing simultaneously, their sink nodes will have data flowing nondeterministically. Such condition models this scenario, considering when both cases may happen as two nondeterministic "distinct" possible executions. Therefore, the operation $acc \circ (X \to Y)) \setminus s'_j$ removes every interpretation of $s'$ which sink node equals $Y$, while $go(t, s', acc)$ denotes an execution containing the removed $s'_j$ but not considering $X \to Y$. The return condition $s = \varepsilon$ denotes that the program as a whole has already been processed.

Considering the cases including block programs induced by SyncDrain and AsyncDrain connectors, $halt(A, B, s')$ is defined as a supporting function that will be used in the case the block program conditions fail. Then, data flow that was in the ports of the SyncDrain/AsyncDrain evaluated cannot be further considered in this execution steps: channels that have their sink node pointed to $A$ or $B$.

Intuitively, *go* is a function that processes a program $\pi$ with input $t$ as the program's data initially available at ports $p \in \pi$ and returns the next data configuration after processing all connectors and verifying whether they are eligible for data to flow. The return of *go* depends on a function $fire$ which is bound to return the final configuration of the Reo circuit after an iteration (i.e., the last ports that data flow). We define $sink(s'_j)$ as the sink node of a connector, in this case, the port name where a data item flowing into a Reo connector is bound to. The operation denoted by $\cup$ is the standard set union.

Definition *go* employs a function named $fire\colon T \times s \to T$ which returns the firing of all possible data flows in the Reo connector, given the Reo program $\pi$ and an initial data flow on ports of $\pi$. The set $T$ is the set of possible data flows as constructed by the BNF grammar in Definition 1. The function $fire$ returns the resulting data flow of this execution step by considering the program processed by *go* as $s$ and the current step's data flow $t$. Parameter $s$ contains *ReLo* programs as yielded by *parse*.

**Definition 6** (Data marking relation $fire$)**.**

$$fire(t,s) = \begin{cases} \varepsilon, \text{ if } s = \varepsilon \\ AxB \circ fire(t,s'), \text{ if } s = (AxB) \circ s' \text{ and } Ax \prec t \\ B(f(a)) \circ fire(t,s'), \text{ if } s = (f(D_A) \to B) \circ s' \text{ and } Ax \prec t \\ Bx \circ fire(t,s'), \text{ if } \begin{cases} s = (A \to B) \circ s' \text{ and } Ax \prec t, or \\ s = (AxB \to Bx) \circ s' \text{ and } axb \prec t \end{cases} \end{cases} \tag{1}$$

We define $f_{ReLo}$ as the transition relation of a *ReLo* model. It denotes how the transitions of the model fire, i.e., given an input $t$ and a program $\pi$ denoting a Reo circuit, $f_{ReLo}(t, \pi)$ interfaces with *go* to return the resulting data flow of $\pi$ given that data depicted by $t$ are flowing in the connector's ports.

**Definition 7.** Transition relation $f_{ReLo}(t, \pi) = go(t, (parse(\pi, [])), [])$

We define $f_{ReLo}(t, \pi^\star)$ as the application of $f_{ReLo}(t, \pi)$ iteratively for the (nondeterministic finite) number of steps denoted by $\star$, starting with $t$ with $\pi$, and considering the obtained intermediate $t'$ in the steps.

A *ReLo* frame is a structure based on Kripke frames [24] formally defined as a tuple $\mathscr{F} = \langle S, \Pi, R_\Pi, \delta, \lambda \rangle$, where each element of $\mathscr{F}$ is described by Definition 8.

**Definition 8** (*ReLo* frame)**.** $S$ is a non-empty enumerable set of states and $\Pi$ a Reo program.

- $R_\Pi \subseteq S \times S$ is a relation defined as follows.
  - $R_{\pi_i} = \{uR_{\pi_i}v \mid f_{ReLo}(t, \pi_i) \prec \delta(v),\ t \prec \delta(u)\}$, $\pi_i$ is any combination of any atomic program which is a subprogram of $\Pi$.
  - $R_{\pi_i^\star} = R^\star_{\pi_i}$, the reflexive transitive closure (RTC) of $R_{\pi_i}$.

- $\lambda : S \times \mathcal{N} \to \mathbb{R}$ is a function that returns the time instant a data item in a data markup flows through a port name of $\mathcal{N}$.
- $\delta : S \to T$, is a function that returns data in ports of the circuit in a state $s \in S$, $T$ being the set of possible data flows in the model.

From Definition 8, a *ReLo* model is formally defined as a tuple $\mathcal{M} = \langle \mathcal{F}, \mathbf{V} \rangle$ by Definition 9. Intuitively, it is a tuple consisting of a *ReLo* frame and a valuation function, which given a state $w$ of the model and a propositional symbol $\varphi \in \Phi$, maps to either $true$ or $false$.

**Definition 9** (*ReLo* models)**.** A model in *ReLo* is a tuple $\mathcal{M} = \langle \mathcal{F}, \mathbf{V} \rangle$, where $\mathcal{F}$ is a *ReLo* frame and $V : S \times \Phi \to \{true, false\}$ is the model's valuation function

**Definition 10** (Satisfaction notion)**.**

- $\mathcal{M}, s \Vdash p$ iff $V(s, p) = true$
- $\mathcal{M}, s \Vdash \top$ always
- $\mathcal{M}, s \Vdash \neg\varphi$ iff $\mathcal{M}, s \nVdash \varphi$
- $\mathcal{M}, s \Vdash \varphi_1 \wedge \varphi_2$ iff $\mathcal{M}, s \Vdash \varphi_1$ and $\mathcal{M}, s \Vdash \varphi_2$
- $\mathcal{M}, s \Vdash \langle t, \pi \rangle \varphi$ if there exists a state $w \in S$, $sR_\pi w$, and $\mathcal{M}, s \Vdash \varphi$

We denote by $\mathcal{M} \Vdash \varphi$ if $\varphi$ is satisfied in all states of $\mathcal{M}$. By $\Vdash \varphi$ we denote that $\varphi$ is valid in any state of any model.

We recover the circuit in Fig. 2 as an example. 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 $V(D_{X\,fifo\,Y}, p) = true$ as $D_{X\,fifo\,Y}$ is the only state where $D_X R_\Pi D_{X\,fifo\,Y}$. For example, one might state $p$ as "There is no port with any data flow", hence $V(D_{X\,fifo\,Y}, p) = true$.

As another usage example, we 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 example state that for this data flow, after every single execution of $\pi$, 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 2.** $[X1, \pi]\neg(D_A = 1 \wedge D_B = 1 \wedge D_C = 1) \wedge t' = X1Y$, where $t' = f_{ReLo}(t, \pi)$
$\mathcal{M}, D_X \Vdash [X1, \pi]\neg(D_A = 1 \wedge D_B = 1 \wedge D_C = 1) \wedge t' = X1Y$.
$\mathcal{M}, D_{x \text{—}\square\!\!\rightarrow y} \Vdash \neg(D_A = 1 \wedge D_B = 1 \wedge D_C = 1) \wedge t' = X1Y$.
$\mathcal{M}, D_{x \text{—}\square\!\!\rightarrow y} \Vdash \neg(D_A = 1 \wedge D_B = 1 \wedge D_C = 1)$ and $\mathcal{M}, D_{x \text{—}\square\!\!\rightarrow y} \Vdash t' = X1Y$.

The notion of $\mathcal{M}, D_X \Vdash \langle t, \pi^\star \rangle p$ holds if a state $s$ is reached from $D_X$ by means of $R^\star_\pi$ with $V(s, p) = \top$. If we state $p$ as "the data item of port $X$ equals 1", it holds because $D_X R^\star_\pi D_X$ and $V(D_X, p) = \top$. If there is an execution of $\pi$ that lasts a nondeterministic finite number of iterations, and there is data in $C$ equals to 1, then there is an execution under the same circumstances where the same data has been in $B$.

**Example 3.** $\langle t, \pi^\star \rangle D_C = 1 \to \langle t, \pi^\star \rangle D_B = 1$
$\mathcal{M}, D_X \Vdash \langle t, \pi^\star \rangle D_C = 1 \to \langle t, \pi^\star \rangle D_B = 1$
$\mathcal{M}, D_X \Vdash \neg(\langle t, \pi^\star \rangle D_C = 1) \vee \langle t, \pi^\star \rangle D_B = 1$
$\mathcal{M}, D_X \Vdash [t, \pi^\star]\neg D_C = 1 \vee \langle t, \pi^\star \rangle D_B = 1$
$\mathcal{M}, D_X \Vdash [t, \pi^\star]\neg D_C = 1$ or $\mathcal{M}, D_X \Vdash \langle t, \pi^\star \rangle D_B = 1$
$\mathcal{M}, D_X \Vdash \langle t, \pi^\star \rangle D_B = 1$, because $\mathcal{M}, D_B \Vdash D_B = 1$ and $D_X R_{\pi^\star} R_B$.

### 4.1 Axiomatic System

We define an axiomatization of *ReLo*, discuss its soundness and completeness.

**Definition 11** (Axiomatic System)**.**

**(PL)** Enough Propositional Logic tautologies
**(K)** $[t,\pi](\varphi \to \psi) \to ([t,\pi]\varphi \to [t,\pi]\psi)$
**(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)** $\langle t,\pi\rangle\varphi \leftrightarrow \varphi$ iff $f_{ReLo}(t,\pi) = \varepsilon$
**(It)** $\varphi \wedge [t,\pi][t_{(f,b)},\pi^\star]\varphi \leftrightarrow [t,\pi^\star]\varphi$, $t_{(f,b)} = f_{ReLo}(t,\pi)$
**(Ind)** $\varphi \wedge [t,\pi^\star](\varphi \to [t_{(f,b)^\star},\pi]\varphi) \to [t,\pi^\star]\varphi$, $t_{(f,b)^\star} = f_{ReLo}(t,\pi^\star)$

$$\textbf{(MP)}\ \frac{\varphi \qquad \varphi \to \psi}{\psi} \qquad \textbf{(Gen)}\ \frac{\varphi}{[t,\pi]\varphi}$$

**Lemma 1** (Soundness)**.** *Proof.*
Axioms **(PL)**, **(K)**, **(And)** and **(Du)** are standard in Modal Logic literature, along with rules **(MP)** and **(Gen)** [15]. Axiom **(It)** and **(Ind)** are similar from PDL. **(R):** $\langle t,\pi\rangle\varphi \leftrightarrow \varphi$ iff $f_{ReLo}(t,\pi) = \varepsilon$
Suppose by contradiction that exists a state $s$ from a model $\mathscr{M} = \langle S,\Pi,R_\Pi,\delta,\lambda,V\rangle$ where **(R)** does not hold. There are two possible cases.
($\Rightarrow$) Suppose by contradiction $\mathscr{M},s \Vdash \langle t,(f,b)\rangle\varphi$ and $\mathscr{M},s \nVdash \varphi$. $\mathscr{M},s \Vdash \langle t,(f,b)\rangle\varphi$ iff there is a state $v \in S$ such that $sR_\pi v$. Because $f_{ReLo}(t,(f,b)) = \varepsilon, s = v$ (i.e., in this execution no other state is reached from $s$). Therefore, $\mathscr{M},s \Vdash \varphi$, contradicting $\mathscr{M},s \nVdash \varphi$.
($\Leftarrow$) Suppose by contradiction $\mathscr{M},s \Vdash \varphi$ and $\mathscr{M},s \nVdash \langle t,(f,b)\rangle\varphi$. In order to $\mathscr{M},s \nVdash \langle t,(f,b)\rangle\varphi$, for every state $v \in S$ such that $sR_\pi v$, $\mathscr{M},v \nVdash \varphi$. Because $f_{ReLo}(t,(f,b)) = \varepsilon, s = v$ (i.e., in this execution no other state is reached from $s$). Therefore, $\mathscr{M},v \nVdash \varphi$, contradicting $\mathscr{M},v \Vdash \varphi$.

□

### 4.2 Completeness

We start by defining the Fisher-Ladner closure of a formula as the set closed by all of its subformulae, following the idea employed in other modal logic works [15, 9] as follows.

**Definition 12** (Fisher-Ladner Closure)**.** Let $\Phi$ be a the set of all formulae in *ReLo*. The Fischer-Ladner closure of a formula, notation $FL(\varphi)$ is inductively defined as follows:

- $FL\colon \Phi \to 2^\Phi$
- $FL_{(f,b)}\colon \{\langle t,(f,b)\rangle\varphi\} \to 2^\Phi$, where $(f,b)$ is a *ReLo* program and $\varphi$ a *ReLo* formula.

These functions are defined as

- $FL(p) = \{p\}$, $p$ an atomic proposition;
- $FL(\varphi \to \psi) = \{\varphi \to \psi\} \cup FL(\varphi) \cup FL(\psi)$
- $FL_{(f,b)}(\langle t,(f,b)\rangle\varphi) = \{\langle t,(f,b)\rangle\varphi\}$
- $FL(\langle t,(f,b)\rangle\varphi) = FL_{(f,b)}((\langle t,(f,b)\rangle\varphi) \cup FL(\varphi)$
- $FL_{(f,b)}(\langle t,(f,b)^\star\rangle\varphi) = \{\langle t,(f,b)^\star\rangle\varphi\} \cup FL_{(f,b)}(\langle t,(f,b)\rangle\langle t,(f,b)^\star\rangle\varphi)$
- $FL(\langle t,(f,b)^\star\rangle\varphi) = FL_{(f,b)}((\langle t,(f,b)^\star\rangle\varphi) \cup FL(\varphi)$

From the definitions above, we prove two lemmas that can be understood as properties that formulae need to satisfy to belong to their Fisher-Ladner closure.

**Lemma 2.** *If* $\langle t,(f,b)\rangle\psi \in FL(\varphi)$*, then* $\psi \in FL(\varphi)$

**Lemma 3.** *If* $\langle t,(f,b)^\star\rangle\psi \in FL(\varphi)$*, then* $\langle t,(f,b)\rangle\langle t,(f,b)^\star\rangle\psi \in FL(\varphi)$

The proofs for Lemmas 2 and 3 are straightforward from Definition 12. The following definitions regard the definitions of maximal canonical subsets of *ReLo* formulae. We first extend Definition 12 to a set of formulae $\Gamma$. The Fisher-Ladner closure of a set of formulae $\Gamma$ is $FL(\Gamma) = \bigcup_{\varphi\in\Gamma} FL(\varphi)$. Therefore, $FL(\Gamma)$ is closed under subformulae. For the remainder of this section, we will assume that $\Gamma$ is finite.

**Lemma 4.** *If* $\Gamma$ *is a finite set of formulae, then* $FL(\Gamma)$ *also is a finite set of formulae*

*Proof.* The proof is standard in literature [10]. Intuitively, because $FL$ is defined recursively over a set of formulae $\Gamma$ into formulae $\psi$ of a formula $\varphi \in \Gamma$, $\Gamma$ being finite leads to the resulting set of $FL(\Gamma)$ also being finite (at some point, all atomic formulae composing $\varphi$ will have been reached by $FL$). □

**Definition 13** (Atom)**.** Let $\Gamma$ be a set of consistent formulae. An atom of $\Gamma$ is a set of formulae $\Gamma'$ that is a maximal consistent subset of $FL(\Gamma)$. The set of all atoms of $\Gamma$ is defined as $At(\Gamma)$.

**Lemma 5.** *Let* $\Gamma$ *a consistent set of formulae and* $\psi$ *a ReLo formula. If* $\psi \in FL(\Gamma)$*, and* $\psi$ *is satisfiable then there is an atom of* $\Gamma$*,* $\Gamma'$ *where* $\psi \in \Gamma'$*.*

*Proof.* The proof follows from Lindembaum's lemma. From Lemma 4, as $FL(\Gamma)$ is a finite set, its elements can be enumerated from $\gamma_1,\gamma_2,\ldots,\gamma_n, n = |FL(\Gamma)|$. The first set, $\Gamma'_1$ contains $\psi$ as the starting point of the construction. Then, for $i = 2,\ldots,n$, $\Gamma'_i$ is the union of $\Gamma'_{i-1}$ with either $\{\gamma_i\}$ or $\{\neg\gamma_i\}$, respectively whether $\Gamma'_i \cup \{\gamma_i\}$ or $\Gamma'_i \cup \{\neg\gamma_i\}$ is consistent. In the end, we make $\Gamma' = \Gamma'_n$ as it contains the union of all $\Gamma_i, 1 \le i \le n$. This is summarized in the following bullets:

- $\Gamma'_1 = \{\psi\}$;
- $\Gamma'_i, = \begin{cases} \Gamma'_{i-1} \cup \{\gamma_i\}, \text{ if } \Gamma_{n-1} \cup \{\gamma_n\} \text{ is consistent} \\ \Gamma'_{i-1} \cup \{\neg\gamma_i\}, \text{ otherwise} \end{cases}$ for $1 < i < n$;
- $\Gamma = \bigcup_{i=1}^{n} \Gamma_i$

□

**Definition 14** (Canonical relations over $\Gamma$)**.** Let $\Gamma$ a set of formulae, $A,B$ atoms of $\Gamma$ ($A,B \in At(\Gamma)$), $\Pi$ a *ReLo* program and $\langle t,(f,b)\rangle\varphi \in At(\Gamma)$. The canonical relations on $At(\Gamma)$ is defined as $S^\Gamma_\Pi$ as follows:
$AS^\Gamma_\Pi B \leftrightarrow \bigwedge A \wedge \langle t,(f,b)\rangle \bigwedge B)$ is consistent , $AS^\Gamma_{\Pi^\star} B \leftrightarrow \bigwedge A \wedge \langle t,(f,b)^\star\rangle \bigwedge B)$ is consistent

Definition 14 states that the relation between two atoms of $\Gamma$, $A$ and $B$ is done by the conjunction of the formulae in $A$ with all formulae in $B$ which can be accessed from $A$ with a diamond formula, such that this conjunction is also a consistent formula. Intuitively, it states that $A$ and $B$ are related in $S^\Gamma_\Pi$ by every formula $\varphi$ of $B$ which conjunction with $A$ by means of a diamond results in a consistent scenario.

The following definition is bound to formalize the canonical version of $\delta$ as the data markup function.

**Definition 15** (Canonical data markup function $\delta^\Gamma_c$)**.**
Let $F = \{\langle t_1,(f_1,b_1)\rangle\varphi_1, \langle t_2,(f_2,b_2)\rangle\varphi_2,\ldots,\langle t_n,(f_n,b_n)\rangle\varphi_n\}$ be the set of all diamond formula occurring on an atom $A$ of $\Gamma$. The canonical data markup is defined as $\delta^\Gamma_c : At(\Gamma) \to T$ as follows:

- The sequence $\{t_1,t_2,\ldots,t_n\} \subseteq \delta(A)$ Therefore, $\{t_1,t_2,\ldots,t_n\} \subseteq \delta^\Gamma_c(A)$. Intuitively, this states that all the data flow in the set of formulae must be valid data markups of $A$, which leads to them to also be valid data markups of $\delta^\Gamma_c$ following Definition 14.
- for all programs $\pi = (f,b) \in \Pi$, $f_{ReLo}((\delta^\Gamma_c(A)),(f,b)) \prec \delta^\Gamma_c(B) \leftrightarrow AS^\Gamma_\Pi B$.

**Definition 16** (Canonical model)**.** A canonical model over a set of formulae $\Gamma$ is defined as a *ReLo* model $\mathscr{M}_c^\Gamma = \langle At(\Gamma), \Pi, S_\Pi^\Gamma, \delta_c^\Gamma, \lambda_c, V_c^\Gamma \rangle$, where:

- $At(\Gamma)$ is the set of states of the canonical model;
- $\Pi$ is the model's *ReLo* program;
- $S_\Pi^\Gamma$ are the canonical relations over $\Gamma$;
- $\delta_c^\Gamma$ is the canonical markup function;
- $\lambda_c \colon At(\Gamma) \times \mathscr{N} \to \mathbb{R}$;
- $V_c^\Gamma \colon At(\Gamma) \times \varphi \to \{true, false\}$, namely $V_c^\Gamma(A,p) = \{A \in At(\Gamma) \mid p \in A\}$;

**Lemma 6.** *For all programs $\pi = (f,b)$ that compose $\Pi$, $t = \delta_c^\Gamma(A)$:*

1. *if $f_{ReLo}(t,(f,b)) \neq \varepsilon$, then $f_{ReLo}(t,(f,b)) \prec \delta_c^\Gamma(B)$ iff $AS_\Pi^\Gamma B$.*
2. *if $f_{ReLo}(t,(f,b)) = \varepsilon$, then $(A,B) \notin S_\Pi^\Gamma$.*

*Proof.* The proof for 1. is straightforward from Definition 15. The proof for 2. follows from axiom *R*. Because $f_{ReLo}(t,(f,b)) = \varepsilon$, no other state is reached from the current state, hence no state *B* related with *A* by $R_\Pi^\Gamma$ can be reached. □

The following lemma states that canonical models always exists if there is a formula $\langle t,(f,b)\varphi\rangle \in FL(\Gamma)$, a set of formulae $\Gamma$ and a Maximal Consistent Set $A \in At(\Gamma)$. This assures that given the required conditions, a canonical model can always be built.

**Lemma 7** (Existence Lemma for canonical models)**.** *Let A be an atom of $At(\Gamma)$ and $\langle t,(f,b)\rangle\varphi \in FL(\Gamma)$. $\langle t,(f,b)\rangle\varphi \in A \iff \exists$ an atom $B \in At(\Gamma)$ such that $AS_\Pi^\Gamma B$, $t \prec \delta_c^\Gamma(A)$ and $\varphi \in B$.*

*Proof.* $\Rightarrow$ Let $A \in At(\Gamma)$ $\langle t,(f,b)\rangle\varphi \in FL(\Gamma)$ and $\langle t,(f,b)\rangle\varphi \in A$ . Because $A \in At(\Gamma)$, from Definition 15 we have $t \prec \delta_c^\Gamma(A)$. From Lemma 5 we have that if $\psi \in FL(\Gamma)$ and $\psi$ is consistent, then there is an atom of $\Gamma$, $\Gamma'$ where $\psi \in \Gamma'$. Rewriting $\varphi$ as $(\varphi \wedge \gamma) \vee (\varphi \wedge \neg\gamma)$ (a tautology from Propositional Logic), an atom $B \in At(\Gamma)$ can be constructed, because either $\langle t,(f,b)\rangle(\varphi \wedge \gamma)$ or $\langle t,(f,b)\rangle(\varphi \wedge \neg\gamma)$ is consistent. Therefore, considering all formulae $\gamma \in FL(\Gamma)$, $B \in At(\Gamma)$ is constructed with $\varphi \in B$ and $A \wedge (\langle t,(f,b)\rangle\varphi \bigwedge B$. From Definition 14, $AS_\Pi^\Gamma B$.
$\Leftarrow$ Let $A \in At(\Gamma)$ and $\langle t,(f,b)\rangle\varphi \in FL(\Gamma)$. Also, let $B \in At(\Gamma)$, $AS_\Pi^\Gamma B$, $t \prec \delta_c^\Gamma(A)$, and $\varphi \in B$. As $AS_\Pi^\Gamma B$, from Definition 14, $AS_\Pi^\Gamma B \leftrightarrow (A \wedge \langle t,(f,b)\rangle \bigwedge B)$, $\forall \varphi_i \in B$ is consistent. From $\varphi \in B$, $(A \wedge \langle t,(f,b)\rangle\varphi)$ is also consistent. As $A \in At(\Gamma)$ and $\langle t,(f,b)\varphi \in FL(\Gamma)$, by Definition 13, as *A* is maximal, then $\langle t,(f,b)\rangle\varphi \in A$.

□

The following lemma formalizes the truth notion for a canonical model $\mathscr{M}_c^\Gamma$, given a state *s* and a formula $\varphi$. It formalizes the semantic notion for canonical models in *ReLo*.

**Lemma 8** (Truth Lemma)**.** *Let $\mathscr{M}_c^\Gamma = \langle At(\Gamma), \Pi, S_\Pi^\Gamma, \delta_c^\Gamma, \lambda, V_c^\Gamma \rangle$ be a canonical model over a formula $\gamma$. Then, for every state $A \in At(\Gamma)$ and every formula $\varphi \in FL(\gamma)$: $\mathscr{M}_c^\Gamma, A \Vdash \varphi \iff \varphi \in A$.*

*Proof.* The proof proceeds by induction over the structure of $\varphi$.

- Induction basis: suppose $\varphi$ is a proposition *p*. Therefore, $\mathscr{M}_c^\Gamma, A \Vdash p$. From Definition 16, $\mathscr{M}_c^\Gamma$'s valuation function is $V_c^\Gamma(p) = \{A \in At(\Gamma) \mid p \in A\}$. Therefore, $p \in A$.
- Induction Hypothesis: Suppose $\varphi$ is a non atomic formula $\psi$. Then, $\mathscr{M}_c^\Gamma, A \Vdash \psi \iff \psi \in A$, $\psi$ a strict subformula of $\varphi$.
- Inductive step: Let us prove it holds for the following cases (we ommit propositional operators):

- Case $\varphi = \langle t,(f,b)\rangle\phi$. Then, $\mathscr{M}_c^\Gamma, A \Vdash \langle t,(f,b)\rangle\phi \iff \langle t,(f,b)\rangle\phi \in A$:
  $\rightarrow$ Let $\mathscr{M}_c^\Gamma, A \Vdash \langle t,(f,b)\rangle\phi$. From Definition 14, there is a state $B$ where $AS_\Pi^\Gamma B$ and $\phi \in B$. By Lemma 7, $\langle t,(f,b)\rangle\phi \in A$. Therefore, it holds.
  $\leftarrow$ Let $\mathscr{M}_c^\Gamma, A \nVdash \langle t,(f,b)\rangle\phi$. From Definition 16's valuation function $V_c^\Gamma$ and Lemma 5, we have $\mathscr{M}_c^\Gamma, A \Vdash \neg\langle t,(f,b)\rangle\phi$. Therefore, for every $B$ where $AS_\Pi^\Gamma B, \mathscr{M}_c^\Gamma, B \Vdash \neg\phi$. From the induction hypothesis, $\phi \notin B$. Hence, From Lemma 7, $\langle t,(f,b)\rangle\phi \notin A$.
- Case $\varphi = \langle t,(f,b)^\star\rangle\phi$. Then, $\mathscr{M}_c^\Gamma, A \Vdash \langle t,(f,b)^\star\rangle\phi \iff \langle t,(f,b)^\star\rangle\phi \in A$:
  $\rightarrow$ Let $\mathscr{M}_c^\Gamma, A \Vdash \langle t,(f,b)^\star\rangle\phi$. From Definition 14, there is a state $B$ where $AS_{\Pi^\star}^\Gamma B$ and $\phi \in B$. By Lemma 7, $\langle t,(f,b)^\star\rangle\phi \in A$. Therefore, it holds.
  $\leftarrow$ Let $\mathscr{M}_c^\Gamma, A \nVdash \langle t,(f,b)^\star\rangle\phi$. From Definition 16's valuation function $V_c^\Gamma$ and Lemma 5, we have $\mathscr{M}_c^\Gamma, A \Vdash \neg\langle t,(f,b)^\star\rangle\phi$. Therefore, for every $B$ where $AS_{\Pi^\star}^\Gamma B, \mathscr{M}_c^\Gamma, B \Vdash \neg\phi$. From the induction hypothesis, $\phi \notin B$. Hence, From Lemma 7, $\langle t,(f,b)^\star\rangle\phi \notin A$.

□

We proceed by formalizing the following lemma, which is bound to show that the properties that define $\star$ for regular *ReLo* models also holds in *ReLo* canonical models.

**Lemma 9.** *Let $A,B \in At(\Gamma)$ and $\Pi$ a ReLo program. If $AS_{\Pi^\star}B$ then $AS_\Pi^\star B$*

*Proof.* Suppose $AS_{\Pi^\star}B$. Define $C = \{C' \in At(\Gamma) \mid AS_\Pi^\star C\}$ as the set of all atoms $C'$ which $A$ reaches by means of $S_{\Pi^\star}$. We will show that $B \in C$. Let $C_c$ be the maximal consistent set obtained by means of Lemma 5, $C_c = \{\bigwedge C_1 \vee C_2 \vee \ldots \bigwedge C_n\}$, where the conjunction of each $C_i$ is consistent, and each $C_i$ is a maximal consistent set. Also, define $t = \delta_c^\Gamma(C_c)$ as the canonical markup of $C_c$.

Note that $C_c \wedge \langle t,(f,b)\rangle\neg C_c$ is inconsistent: if it was consistent, then for some $D \in At(\Gamma)$ which $A$ cannot reach, $C_c \wedge \langle t,(f,b)\rangle \bigwedge D$ would be consistent, which leads to $\bigwedge C_1 \vee C_2 \vee \cdots \vee C_i \vee \langle t,(f,b)\rangle \bigwedge D$ also being consistent, for some $C_i$. By the definition of $C_c$, this means that $D \in C$ but that is not the case (because $D \in C_c$ contradicts $D$ not being reached from $A$ and consequently $C_c$'s definition, as $D \in C_c$ leads to D being reachable from $A$). Following a similar reasoning, $\bigwedge A \wedge \langle t,(f,b)\rangle C_c$ is also inconsistent and therefore its negation, $\bigwedge \neg(A \wedge \langle t,(f,b)\rangle C_c)$ is consistent, which can be rewritten as $\bigwedge A \rightarrow [t,(f,b)]C_c$.

Because $C_c \wedge \langle t,(f,b)\rangle\neg C_c$ is inconsistent, its negation $\neg(C_c \wedge \langle t,(f,b)\rangle\neg C_c)$ is valid, which can be rewritten to $\vdash C_c \rightarrow [t,(f,b)]C_c$ (I). Therefore, by applying generalization we have $\vdash [t,(f,b)^\star](C_c \rightarrow [t,(f,b)]C_c)$. By axiom **(It)**, we derive $\vdash [t,(f,b)]C_c \rightarrow [t,(f,b)^\star]C_c$ (II). By rewriting (II) in (I) we derive $C_c \rightarrow [t,(f,b)^\star]C_c$. As $\bigwedge A \rightarrow [t,(f,b)]C_c$ is valid, from (II) $\bigwedge A \rightarrow [t,(f,b)^\star]C_c$ also is valid. From the hypothesis $AS_{\pi^\star}B$ and $C_c$'s definition, $\bigwedge A \wedge \langle t,(f,b)^\star\rangle B$ and $\bigwedge B \wedge C_c$ are consistent (the latter from $C_c$'s definition). Then, there is a $C_i \in C_c$ such that $\bigwedge B \wedge \bigwedge C$ is consistent. But because each $C_i$ is a maximal consistent set, it is the case that $B = C_i$, which by the definition of $C_c$ leads to $AS_\Pi^\star B$.

□

**Definition 17** (Proper Canonical Model)**.** The proper canonical model over a set of formulae $\Gamma$ is defined as a tuple $\langle At(\Gamma), \Pi, R_\Pi^\Gamma, \delta_\Pi^\Gamma, \lambda_c, V_\Pi^\Gamma\rangle$ as follows:

- $At(\Gamma)$ as the set of atoms of $\Gamma$;
- $\Pi$ as the *ReLo* program;
- The relation $R$ of a *ReLo* program $\Pi$ is inductively defined as:
  - $R_\pi = S_\pi$ for each canonical program $\pi$;
  - $R_{\Pi^\star}^\Gamma = (R_\Pi^\Gamma)^\star$;
  - $\Pi = \pi_1 \odot \pi_2 \odot \cdots \odot \pi_n$ a *ReLo* program, $R_\Pi \subseteq S \times S$ as follows:

  - $R_{\pi_i} = \{uR_{\pi_i}v \mid f_{ReLo}(t,\pi_i) \prec \delta(v)\}$, $t \prec \delta(u)$ and $\pi_i$ is any combination of any atomic programs which is a subprogram of $\Pi$.
- $\delta^{\Gamma}_{\Pi}$ as the canonical markup function;
- $\lambda_c$: $At(\Gamma) \times \mathscr{N} \to \mathbb{R}$;
- $V^{\Gamma}_c(A,p) = \{A \in At(\Gamma) \mid p \in A\}$ as the canonical valuation introduced by Definition 16.

**Lemma 10.** *Every canonical model for* $\Pi$ *has a corresponding proper canonical model: for all programs* $\Pi$, $S^{\Gamma}_{\Pi} \subseteq R^{\Gamma}_{\Pi}$

*Proof.* The proof proceeds by induction on $\Pi$'s length

- For basic programs $\pi$, it follows from Definition 17:
- $\Pi^{\star}$: From Definition 8, $R_{\pi^{\star}} = R^{\star}_{\pi}$. By the induction hypothesis, $S^{\Gamma}_{\Pi} \subseteq R^{\Gamma}_{\Pi}$, Also from the definition of RTC, we have that if $(S^{\Gamma}_{\Pi}) \subseteq (R^{\Gamma}_{\Pi})$, then $(S^{\Gamma}_{\Pi})^{\star} \subseteq (R^{\Gamma}_{\pi})^{\star}$ (i). From Lemma 9, $S^{\Gamma}_{\Pi^{\star}} \subseteq (S^{\Gamma}_{\Pi})^{\star}$, which leads to $(S^{\Gamma}_{\Pi})^{\star} \subseteq (R^{\Gamma}_{\Pi})^{\star}$ by (i). Finally, $(R^{\Gamma}_{\Pi})^{\star} = (R^{\Gamma}_{\Pi^{\star}})$. Hence, $(S^{\Gamma}_{\Pi^{\star}}) \subseteq (R^{\Gamma}_{\Pi^{\star}})$

□

**Lemma 11** (Existence Lemma for Proper Canonical Models)**.** *Let* $A \in At(\Gamma)$ *and* $\langle t,(f,b)\rangle\varphi \in FL(\Gamma)$. *Then,* $\langle t,(f,b)\rangle\varphi \in A \leftrightarrow$ *exists* $B \in At(\Gamma), AR^{\Gamma}_{\Pi}B, t \prec \delta^{\Gamma}_{c}(A)$ *and* $\varphi \in B$.

*Proof.* $\Rightarrow$ Let $\langle t,(f,b)\rangle\varphi \in A$. From Lemma 7 (Existence Lemma for canonical models), There is an atom $B \in At(\Gamma)$ where $AS^{\Gamma}_{\Pi}B$, $t \prec \delta^{\Gamma}_{c}(A)$ and $\varphi \in B$. From Lemma 10, $S^{\Gamma}_{\Pi} \subseteq R^{\Gamma}_{\Pi}$. Therefore, there is an atom $B \in At(\Gamma)$ where $AR^{\Gamma}_{\Pi}B$, $t \prec \delta^{\Gamma}_{c}(A)$ and $\varphi \in B$.
$\Leftarrow$ Let $B$ an atom, $B \in At(\Gamma), AR_{\Pi}B, t \prec \delta^{\Gamma}_{c}(A)$ and $\varphi \in B$. The proof follows by induction on the program $\Pi = (f,b)$ as follows:

- a canonical program $\pi_i$: this case is straightforward as from Definition 17, $S_{\pi_i} = R_{\pi_i}$, and consequently $AS_{\pi_i}B, t \prec \delta^{\Gamma}_{c}(A)$ and (i) $\varphi \in B$. From Lemma 7 and (i), $\langle t,(f,b)\rangle\varphi \in A$.
- $\Pi^{\star}$: from Definition 17, $R_{\Pi^{\star}} = R^{\star}_{\Pi}$. Then, let $B \in At(\Gamma), AR_{\Pi^{\star}}B, t \prec \delta^{\Gamma}_{c}(A)$ and $\varphi \in B$. This means that there is a finite nondeterministic number $n$ where $AR_{\Pi^{\star}}B = AR_{\Pi}A_1R_{\Pi}A_2 \ldots R_{\Pi}A_n$, where $A_n = B$. The proof proceeds by induction on $n$:
  - $n = 1$: $AR_{\Pi}B$ and $\varphi \in B$. Therefore, from Lemma 7,$\langle t,(f,b)\rangle\varphi \in A$. From axiom Rec, one may derive $\Vdash \langle t,(f,b)\rangle\varphi \to \langle t,(f,b)^{\star}\rangle\varphi$. By the definition of FL and $A$'s maximality (as it is an atom of $\Gamma$) $\langle t,(f,b)^{\star}\rangle\varphi \in A$.
  - $n > 1$: From the previous proof step and the induction hypothesis, $\langle t,(f,b)^{\star}\rangle \in A_2$ and $\langle t,(f,b)\rangle\langle t,(f,b)^{\star}\rangle \in A_1$. From axiom Rec, one can derive
    $\Vdash \langle t,(f,b)\rangle\langle t,(f,b)^{\star}\rangle\varphi \to \langle t,(f,b)^{\star}\rangle\varphi$. By the definition of $FL$, and $A$'s maximality (as it is an atom of $\Gamma$), $\langle t,(f,b)^{\star}\rangle\varphi \in A$.

□

**Lemma 12** (Truth Lemma for Proper Canonical Models)**.** *Let* $\mathscr{M}^{\Gamma}_{c} = \langle At(\Gamma), \Pi, R^{\Gamma}_{\Pi}, \delta^{\Gamma}_{\Pi}, \lambda_c, V^{\Gamma}_{\Pi}\rangle$ *a proper canonical model constructed over a formula* $\gamma$. *For all atoms* $A$ *and all* $\varphi \in FL(\gamma)$. $\mathscr{M}, A \Vdash \varphi \leftrightarrow \varphi \in A$.

*Proof.* The proof proceeds by induction over $\varphi$.

- induction basis: $\varphi$ is a proposition p. Therefore, $\mathscr{M}^{\Gamma}_{c}, A \Vdash p$ holds from Definition 17 as $V^{\Gamma}_{c}(p) = \{A \in At(\Gamma) \mid p \in A\}$.
- induction hypothesis: suppose $\varphi$ is a non atomic formula $\psi$. Then, $\mathscr{M}, A \Vdash \varphi \iff \varphi \in A$, $\psi$ a strict subformula of $\varphi$.

- Inductive step: Let us prove it holds for the following cases (we show only for modal cases):
  - Case $\varphi = \langle t,(f,b)\rangle\phi$. Then, $\mathscr{M}_c^\Gamma, A \Vdash \langle t,(f,b)\rangle\phi \iff \langle t,(f,b)\rangle\phi \in A$:
    $\rightarrow$ Let $\mathscr{M}_c^\Gamma, A \Vdash \langle t,(f,b)\rangle\phi$. From Definition 14, there is an atom $B$ where $AS_\Pi^\Gamma B$ and $\phi \in B$. By Lemma 11, $\langle t,(f,b)\rangle\phi \in A$. Therefore, it holds.
    $\leftarrow$
    Let $\mathscr{M}_c^\Gamma, A \nVdash \langle t,(f,b)\rangle\phi$. From Definition 16's valuation function $V_c^\Gamma$ and Lemma 5, we have $\mathscr{M}_c^\Gamma, A \Vdash \neg\langle t,(f,b)\rangle\phi$. Therefore, for every $B$ where $AS_\Pi^\Gamma B, \mathscr{M}_c^\Gamma \Vdash \neg\phi$. From the induction hypothesis, $\phi \notin B$. Hence, from Lemma 11 $\langle t,(f,b)\rangle\phi \notin A$.
  - Case $\varphi = \langle t,(f,b)^\star\rangle\phi$. Then, $\mathscr{M}_c^\Gamma, A \Vdash \langle t,(f,b)^\star\rangle\phi \iff \langle t,(f,b)^\star\rangle\phi \in A$:
    $\rightarrow$ Let $\mathscr{M}_c^\Gamma, A \Vdash \langle t,(f,b)^\star\rangle\phi$. From Definition 14, there is a state $B$ where $AS_{\Pi^\star}^\Gamma B$ and $\phi \in B$. By Lemma 7, $\langle t,(f,b)^\star\rangle\phi \in A$. Therefore, it holds.
    $\leftarrow$ Let $\mathscr{M}_c^\Gamma, A \nVdash \langle t,(f,b)^\star\rangle\phi$. From Definition 16's valuation function $V_c^\Gamma$ and Lemma 5, we have $\mathscr{M}_c^\Gamma, A \Vdash \neg\langle t,(f,b)^\star\rangle\phi$. Therefore, for every $B$ where $AS_{\Pi^\star}^\Gamma B, \mathscr{M}_c^\Gamma, B \Vdash \neg\phi$. From the induction hypothesis, $\phi \notin B$. Hence, From Lemma 7, $\langle t,(f,b)^\star\rangle\phi \notin A$.

□

**Theorem 1** (Completeness of *ReLo*)**.** *Proof.* For every consistent formula $A$, a canonical model $\mathscr{M}$ can be constructed. From Lemma 5, there is an atom $A' \in At(A)$ with $A \in A'$, and from Lemma 12, $\mathscr{M}, A' \Vdash A$. Therefore, *ReLo*'s modal system is complete with respect to the class of proper canonical models as Definition 17 proposes. □

## 5 Conclusions and Further Work

Reo is a widely used tool to model new systems out of the coordination of already existing pieces of software. It has been used in a variety of domains, drawing the attention of researchers from different locations around the world. This has resulted in Reo having many formal semantics proposed, each one employing different formalisms: operational, co-algebraic, and coloring semantics are some of the types of semantics proposed for Reo.

This work extends *ReLo*, a dynamic logic to reason about Reo models. We have discussed its core definitions, syntax, semantic notion, providing soundness and completeness proofs for it. *ReLo* naturally subsumes the notion of Reo programs and models in its syntax and semantics, and implementing its core concepts in Coq enables the usage of Coq's proof apparatus to reason over Reo models with *ReLo*.

Future work may consider the integration of the current implementation of *ReLo* with ReoXplore[4], a platform conceived to reason about Reo models, and extensions to other Reo semantics. Investigations and the development of calculi for *ReLo* are also considered for future work.

## References

[1] JR Abrial (1991): *B-Tool Reference Manual. B-Core (UK) Ltd.*

[2] Farhad Arbab (2004): *Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science* 14(3), p. 329–366.

[3] Farhad Arbab (2006): *Coordination for Component Composition. Electronic Notes in Theoretical Computer Science* 160, pp. 15 – 40. Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005).

[4] https://github.com/frame-lab/ReoXplore2

[4] Farhad Arbab, Natallia Kokash & Sun Meng (2008): *Towards using reo for compliance-aware business process modeling*. In: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, Springer, pp. 108–123.

[5] Farhad Arbab & Jan JMM Rutten (2002): *A coinductive calculus of component connectors*. In: *International Workshop on Algebraic Development Techniques*, Springer, pp. 34–55.

[6] Colin Atkinson & Thomas Kuhne (2003): *Model-driven development: a metamodeling foundation*. *IEEE software* 20(5), pp. 36–41.

[7] Christel Baier (2005): *Probabilistic Models for Reo Connector Circuits. J. UCS* 11(10), pp. 1718–1748.

[8] Christel Baier, Marjan Sirjani, Farhad Arbab & Jan Rutten (2006): *Modeling component connectors in Reo by constraint automata*. *Science of computer programming* 61(2), pp. 75–113.

[9] Mario Benevides, Bruno Lopes & Edward Hermann Haeusler (2018): *Towards reasoning about Petri nets: A Propositional Dynamic Logic based approach*. *Theoretical Computer Science* 744, pp. 22–36.

[10] Patrick Blackburn, M De Rijke & Y Venema (2001): *Cambridge tracts in theoretical computer science*.

[11] Roberto Bruni & Ugo Montanari (2000): *Zero-safe nets: Comparing the collective and individual token approaches*. *Information and computation* 156(1-2), pp. 46–89.

[12] Dave Clarke (2007): *Coordination: Reo, nets, and logic*. In: *International Symposium on Formal Methods for Components and Objects*, Springer, pp. 226–256.

[13] Erick Grilo & Bruno Lopes (2020): *ReLo: a dynamic logic to reason about Reo circuits1*. In: *Pre-Proceedings of the 15th International Workshop on Logical and Semantic Frameworks, with Applications (LSFA)*, p. 32.

[14] Erick Grilo, Daniel Toledo & Bruno Lopes (2022): *A logical framework to reason about Reo circuits*. *Journal of Applied Logics* 9, pp. 199–254.

[15] David Harel, Dexter Kozen & Jerzy Tiuryn (2001): *Dynamic logic*. In: *Handbook of philosophical logic*, Springer, pp. 99–217.

[16] Daniel Jackson (2002): *Alloy: a lightweight object modelling notation*. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11(2), pp. 256–290.

[17] Sung-Shik TQ Jongmans & Farhad Arbab (2012): *Overview of Thirty Semantic Formalisms for Reo. Scientific Annals of Computer Science* 22(1).

[18] Joachim Klein, Sascha Klüppelholz, Andries Stam & Christel Baier (2011): *Hierarchical modeling and formal verification. An industrial case study using Reo and Vereofy*. In: *International Workshop on Formal Methods for Industrial Critical Systems*, Springer, pp. 228–243.

[19] John C Knight (2002): *Safety critical systems: challenges and directions*. In: *Proceedings of the 24th International Conference on Software Engineering*, ACM, pp. 547–550.

[20] Natallia Kokash & Farhad Arbab (2011): *Formal design and verification of long-running transactions with extensible coordination tools*. *IEEE Transactions on Services Computing* 6(2), pp. 186–200.

[21] Natallia Kokash, Behnaz Changizi & Farhad Arbab (2010): *A semantic model for service composition with coordination time delays*. In: *International Conference on Formal Engineering Methods*, Springer, pp. 106–121.

[22] Natallia Kokash, Christian Krause & Erik De Vink (2012): *Reo+ mCRL2: A framework for model-checking dataflow in service compositions*. *Formal Aspects of Computing* 24(2), pp. 187–216.

[23] Natallia Kokash, Christian Krause & Erik P de Vink (2010): *Data-aware design and verification of service compositions with Reo and mCRL2*. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*, pp. 2406–2413.

[24] Saul A Kripke (1959): *A completeness theorem in modal logic*. *The journal of symbolic logic* 24(1), pp. 1–14.

[25] Yi Li & Meng Sun (2015): *Modeling and verification of component connectors in Coq*. *Science of Computer Programming* 113, pp. 285–301.

[26] Yi Li, Xiyue Zhang, Yuanyi Ji & Meng Sun (2017): *Capturing Stochastic and Real-Time Behavior in Reo Connectors*. In: *Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings*, pp. 287–304, doi:10.1007/978-3-319-70848-5_18.

[27] Yi Li, Xiyue Zhang, Yuanyi Ji & Meng Sun (2019): *A Formal Framework Capturing Real-Time and Stochastic Behavior in Connectors*. *Science of Computer Programming*.

[28] Mohammad Reza Mousavi, Marjan Sirjani & Farhad Arbab (2006): *Formal semantics and analysis of component connectors in Reo*. *Electronic Notes in Theoretical Computer Science* 154(1), pp. 83–99.

[29] M. Saqib Nawaz & Meng Sun (2018): *Reo2PVS: Formal Specification and Verification of Component Connectors*. In: *The 30th International Conference on Software Engineering and Knowledge Engineering, Hotel Pullman, Redwood City, California, USA, July 1-3, 2018.*, pp. 391–390, doi:10.18293/SEKE2018-024.

[30] Jonathan S Ostro (1992): *Formal methods for the specification and design of real-time safety critical systems*. *Journal of Systems and Software* 18(1), pp. 33–60.

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

[32] Bahman Pourvatan, Marjan Sirjani, Hossein Hojjat & Farhad Arbab (2009): *Automated analysis of Reo circuits using symbolic execution*. *Electronic Notes in Theoretical Computer Science* 255, pp. 137–158.

[33] Meng Sun & Yi Li (2014): *Formal modeling and verification of complex interactions in e-government applications*. In: *Proceedings of the 8th International Conference on Theory and Practice of Electronic Governance*, ACM, pp. 506–507.

[34] Samira Tasharofi & Marjan Sirjani (2009): *Formal modeling and conformance validation for WS-CDL using Reo and CASM*. *Electronic Notes in Theoretical Computer Science* 229(2), pp. 155–174.

[35] Xiyue Zhang, Weijiang Hong, Yi Li & Meng Sun (2016): *Reasoning about connectors in Coq*. In: *International Workshop on Formal Aspects of Component Software*, Springer, pp. 172–190.

[36] Xiyue Zhang, Weijiang Hong, Yi Li & Meng Sun (2019): *Reasoning about connectors using Coq and Z3*. *Science of Computer Programming* 170, pp. 27–44.

# Analyzing Innermost Runtime Complexity Through Tuple Interpretations

Liye Guo*
Institute for Computing and Information Sciences
Radboud University, The Netherlands
l.guo@cs.ru.nl

Deivid Vale*
Institute for Computing and Information Sciences
Radboud University, The Netherlands
deividvale@cs.ru.nl

Tuple interpretations are a class of algebraic interpretations that subsume both polynomial and matrix interpretations as they do not impose simple termination and allow non-linearity. They were developed in the context of higher-order rewriting to study derivational complexity of algebraic functional systems. In this paper, we study innermost runtime complexity of first-order applicative rewriting systems by tailoring tuple interpretations to deal with innermost runtime complexity. This simplifies the search for cost interpretations since the strong monotonicity requirement, which is present in full rewriting, is dropped. We prove an innermost version of the compatibility theorem, i.e., if all rules of the system can be oriented, then the complexity relation is contained in the strict (cost) component of the cost–size algebra. We then go on to demonstrate the expressivity of cost–size tuples and how they can be used to bound the runtime complexity of applicative systems.

## 1 Introduction

In the step-by-step computational model induced by rewriting, time complexity is naturally understood as the number of rewriting steps needed to reach normal forms. It is usually the case that the cost of firing a redex, i.e., performing a computational step, is assumed constant. So the intricacies of a low level rewriting realization (for instance in Turing Machines) are ignored. This assumption does not pose a problem as long as the low level time complexity needed to apply a rule is kept low. Additionally, this abstract approach has the advantages of being independent of the specific hardware platform evaluating the rewriting system at hand.

In this rewriting setting, a complexity function bounds the length of rewrite sequences and is parametrized by the size of the starting term of the derivation. Two distinct complexity notions are commonly considered in the literature: derivational and runtime complexity, and they differ by the restrictions imposed on the initial term of derivations. On the one hand, derivational complexity imposes no restriction on the set of initial terms. Intuitively, it captures the worst-case behavior of reducing a term to normal form. On the other hand, runtime complexity requires basic initial terms which, conceptually, are terms where a single function call is performed on data (e.g., integers, lists, and trees) as arguments.

If programs are expressed by rewriting, their execution time is closely related to the runtime complexity of the associated rewrite system. Similarly related are programs using call-by-value evaluation strategy and innermost rewrite systems. Therefore, by combining these two concepts, we obtain a connection between cost analysis of call-by-value programs and the runtime complexity analysis of innermost term rewriting. More importantly, due to the abstract nature of rewriting, it is feasible to forgo any specific

*The authors are supported by the NWO VIDI project "CHORPE", NWO VI.Vidi.193.075 and the NWO TOP project "ICHOR", NWO 612.001.803/7571.

programming language detail and still derive useful term rewriting results that may carry over to programs. For an overview of the applicability of rewriting to program complexity the reader is referred to [1, 19].

Therefore, a rewriting approach to program complexity allows us to fully concentrate on finding techniques to establish bounds to the derivational or runtime complexity functions. A natural way to determine these bounds is adapting the proof techniques used to show termination to deduce the complexity naturally induced by the method. There is a myriad of works following this program. To mention a few, see [2, 4, 6, 13, 14, 20] for interpretation methods, [5, 12, 23] for lexicographic and path orders, and [11, 21] for dependency pairs. In this paper, we follow the same idea and concentrate on investigating the innermost runtime complexity for applicative systems. The termination method for which we base our complexity analysis framework upon is tuple interpretations [16].

Tuple interpretations are an instance of the interpretation method. Thus, we seek to interpret terms in such a way that the rewrite relation can be embedded in a well-founded ordering. The defining characteristic of tuple interpretations is to allow for a split of the complexity measure into abstract notions of cost and size. When distilled into its essence, the ingredient we need to express the concepts of cost and size is: a product $\mathcal{C} \times \mathcal{S}$ of a well-founded set $\mathcal{C}$ — the cost set — and a quasi-ordered set $\mathcal{S}$ — the size set. Intuitively, the cost tuples in $\mathcal{C}$ bound the number of rewriting steps needed to reach normal forms, which is in line with the aforementioned rewriting cost model. Meanwhile, the size tuples in $\mathcal{S}$ are more general. We can use integers, reals, and terms themselves as size. Following the treatment in [16], the construction of cost–size products is done inductively on the structure of types. So we map each type $\sigma$ to a cost–size product $\mathcal{C}_\sigma \times \mathcal{S}_\sigma$. Hence, in this paper our first-order term formalism follows a type discipline.

While forging new tools for our complexity framework, we would like to not only exhibit bounds to the runtime complexity function but also determine sufficient conditions for its feasibility, that is, the existence of polynomial upper bounds. In the eighties Huet and Oppen [15] conjectured that polynomial interpretations are sufficient to evince feasibility, which was disproved by Lautemann [17] in the same decade. In fact, polynomial interpretations induce a double exponential upper bound on the derivation length, as shown by the seminal work of Hofbauer and Lautemann [14]. Feasibility can be recovered by imposing additional conditions on interpretations. To the best of our knowledge, Cichon and Lescanne [6] were the first to propose such conditions even though their setting is restricted to number theoretic functions only. Similar results are proved in [4], where the authors provide rewriting characterizations of complexity classes using bounds to the interpretation of data constructors. These same conditions appear in the higher-order setting, see [2, 16]. In the present paper, we follow a similar approach to that in [4] and show that we can recover those classical results by bounding size-tuples in interpretations.

Tuple interpretations do not provide a complete termination proof method: there are terminating systems for which interpretations cannot be found. Consequently, it does not induce a complete complexity analysis framework either. Notwithstanding, it has the potential to be very powerful if we choose the cost–size sets wisely. A second limitation is that the search for interpretations is undecidable in general, which is expected already in the polynomial case [18]. Undecidability never hindered computer scientists' efforts on mechanizing difficult problems, however. Indeed, several proof search methods were developed over the years to find interpretations automatically [3, 7, 8, 13, 24].

**Contribution.** In Definition 1 we provide a formal definition of cost–size products and interpret types, Definition 3, as cost–size products, which defines the interpretation domain for cost–size algebras defined in Definition 6. In Lemmas 2 and 4 we show the soundness of this approach. In Definition 5 we introduce a type-safe application operator on cost–size products and prove its strong monotonicity, an important ingredient to show the Compatibility Theorem 1. We establish termination of Toyama's system

in Example 3, showing that Theorem 1 correctly captures innermost termination in our setting. We provide sufficient conditions so that feasible bounds on innermost runtime complexity can be achieved in Lemmas 7 and 8.

**Outline.** In Section 2, we fix notation and recall basic notions of rewriting syntax, basic terminology on complexity of rewriting, and review our notation for sets, orders, and functions. In Section 3, we tailor tuple interpretations to the innermost setting and prove the innermost version of the compatibility theorem. We proceed to establish complexity bounds to innermost runtime complexity in Section 4. In Section 5, we present preliminary work on automation techniques to find cost–size tuple interpretations. We conclude the paper in Section 6.

## 2 Preliminaries

**TRSs and Innermost Rewriting.** We consider simply typed first-order term rewriting systems in curried notation. Fix a set $\mathcal{B}$, whose elements are called *sorts*. The set $\mathcal{T}_\mathcal{B}$ of *types* is generated by the grammar $\mathcal{T}_\mathcal{B} ::= \mathcal{B} \mid \mathcal{B} \Rightarrow \mathcal{T}_\mathcal{B}$. Each type is written as $\iota_1 \Rightarrow \cdots \Rightarrow \iota_m \Rightarrow \kappa$ where all $\iota_i$ and $\kappa$ are sorts. A *signature* is a set $\mathcal{F}$ of symbols together with an arity function $\mathtt{ar}$ which associates to each $\mathsf{f} \in \mathcal{F}$ a type $\sigma \in \mathcal{T}_\mathcal{B}$. We call the triple $(\mathcal{B}, \mathcal{F}, \mathtt{ar})$ a *syntax signature*. For each sort $\iota$, we postulate a set $\mathcal{X}_\iota$ of countably many variables and assume that $\mathcal{X}_\iota \cap \mathcal{X}_{\iota'} = \emptyset$ if $\iota \neq \iota'$. Let $\mathcal{X}$ denote $\bigcup_\iota \mathcal{X}_\iota$ and assume that $\mathcal{F} \cap \mathcal{X} = \emptyset$.

The set $\mathbb{T}$ of *pre-terms* is generated by the grammar $\mathbb{T} ::= \mathcal{F} \mid \mathcal{X} \mid (\mathbb{T}\ \mathbb{T})$. The set $T(\mathcal{F}, \mathcal{X})$ of *terms* consists of pre-terms which can be typed as follows: (i) $\mathsf{f} : \sigma$ if $\mathtt{ar}(\mathsf{f}) = \sigma$, (ii) $x : \iota$ if $x \in \mathcal{X}_\iota$, and (iii) $(s\ t) : \tau$ if $s : \iota \Rightarrow \tau$ and $t : \iota$. Application of terms is left-associative, so we write $s\ t\ u$ for $((s\ t)\ u)$. Let $\mathtt{vars}(s)$ be the set of variables occurring in $s$. A *ground term* is a term $s$ such that $\mathtt{vars}(s) = \emptyset$. A symbol $\mathsf{f} \in \mathcal{F}$ is called the *head symbol* of $s$ if $s = \mathsf{f}\ s_1 \ldots s_k$. A *subterm* of $s$ is a term $t$ (we write $s \trianglerighteq t$) such that (i) $s = t$, or (ii) $t$ is a subterm of $s'$ or $s''$ when $s = s'\ s''$. A *proper subterm* of $s$ is a subterm of $s$ which is not equal to $s$. A *substitution* $\gamma$ is a type-preserving map from variables to terms such that the set $\mathtt{dom}(\gamma) = \{x \in \mathcal{X} \mid \gamma(x) \neq x\}$ is finite. Every substitution $\gamma$ extends to a type-preserving map from terms to terms, whose image on $s$ is written as $s\gamma$, as follows: (i) $\mathsf{f}\gamma = \mathsf{f}$, (ii) $x\gamma = \gamma(x)$, and (iii) $(s\ t)\gamma = (s\gamma)\ (t\gamma)$.

A relation $\to$ on terms is *monotonic* if $s \to s'$ implies $t\ s \to t\ s'$ and $s\ u \to s'\ u$ for all terms $t$ and $u$ of appropriate types. A *rewrite rule* $\ell \to r$ is a pair of terms of the same type such that $\ell = \mathsf{f}\ \ell_1 \ldots \ell_k$ and $\mathtt{vars}(\ell) \supseteq \mathtt{vars}(r)$. A *term rewriting system* (TRS) $\mathcal{R}$ is a set of rewrite rules. The *rewrite relation* $\to_\mathcal{R}$ induced by $\mathcal{R}$ is the smallest monotonic relation on terms such that $\ell\gamma \to_\mathcal{R} r\gamma$ for all rules $\ell \to r \in \mathcal{R}$ and substitutions $\gamma$. A *reducible expression* (redex) is a term of form $\ell\gamma$ for some rule $\ell \to r$ and substitution $\gamma$. A term is in *normal form* if none of its subterms is a redex. A TRS $\mathcal{R}$ is *terminating* if no infinite rewrite sequence $s \to_\mathcal{R} s' \to_\mathcal{R} s'' \to_\mathcal{R} \cdots$ exists.

Every rewrite rule $\ell \to r$ *defines* a symbol $\mathsf{f}$, namely, the head symbol of $\ell$. For each $\mathsf{f} \in \mathcal{F}$, let $\mathcal{R}_\mathsf{f}$ denote the set of rewrite rules that define $\mathsf{f}$ in $\mathcal{R}$. A symbol $\mathsf{f} \in \mathcal{F}$ is a *defined symbol* if $\mathcal{R}_\mathsf{f} \neq \emptyset$; otherwise, $\mathsf{f}$ is called a *constructor*. Let $\mathcal{D}$ be the set of defined symbols and $\mathcal{C}$ the set of constructors. So $\mathcal{F} = \mathcal{D} \cup \mathcal{C}$. A *data term* is a term of form $\mathsf{c}\ d_1\ \ldots\ d_k$ where $\mathsf{c}$ is a constructor and each $d_i$ is a data term. A *basic term* is a term of type $\iota$ and of form $\mathsf{f}\ d_1\ \ldots\ d_m$ where $\iota$ is a sort, $\mathsf{f}$ is a defined symbol and all $d_1, \ldots, d_m$ are data terms. We let $T_b(\mathcal{F})$ denote the set of all basic terms.

**Example 1** We fix $\mathsf{nat}$ and $\mathsf{list}$ for the sorts of natural numbers and lists of natural numbers, respectively. In the below TRS, $\mathsf{0} : \mathsf{nat}$, $\mathsf{s} : \mathsf{nat} \Rightarrow \mathsf{nat}$, $\mathsf{nil} : \mathsf{list}$ and $\mathsf{cons} : \mathsf{nat} \Rightarrow \mathsf{list} \Rightarrow \mathsf{list}$ are constructors while $\mathtt{add}, \mathtt{minus}, \mathtt{quot} : \mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}$, $\mathtt{append} : \mathsf{list} \Rightarrow \mathsf{list} \Rightarrow \mathsf{list}$, $\mathtt{sum} : \mathsf{list} \Rightarrow \mathsf{nat}$ and $\mathtt{rev} : \mathsf{list} \Rightarrow \mathsf{list}$ are

defined symbols.

$$\begin{array}{rlrl}
\texttt{add}\ x\ \mathsf{0} &\to x & \texttt{sum}\ \mathsf{nil} &\to \mathsf{0}\\
\texttt{add}\ x\ (\mathsf{s}\ y) &\to \mathsf{s}\ (\texttt{add}\ x\ y) & \texttt{sum}\ (\mathsf{cons}\ x\ q) &\to \texttt{add}\ (\texttt{sum}\ q)\ x\\
\texttt{append}\ \mathsf{nil}\ l &\to l & \texttt{rev}\ \mathsf{nil} &\to \mathsf{nil}\\
\texttt{append}\ (\mathsf{cons}\ x\ q)\ l &\to \mathsf{cons}\ x\ (\texttt{append}\ q\ l) & \texttt{rev}\ (\mathsf{cons}\ x\ q) &\to \texttt{append}\ (\texttt{rev}\ q)\ (\mathsf{cons}\ x\ \mathsf{nil})\\
\mathsf{minus}\ x\ \mathsf{0} &\to x & \mathsf{quot}\ \mathsf{0}\ (\mathsf{s}\ y) &\to \mathsf{0}\\
\mathsf{minus}\ \mathsf{0}\ y &\to \mathsf{0} & \mathsf{quot}\ (\mathsf{s}\ x)\ (\mathsf{s}\ y) &\to \mathsf{s}\ (\mathsf{quot}\ (\mathsf{minus}\ x\ y)\ (\mathsf{s}\ y))\\
\mathsf{minus}\ (\mathsf{s}\ x)\ (\mathsf{s}\ y) &\to \mathsf{minus}\ x\ y
\end{array}$$

We restrict our attention to innermost rewriting: only redexes with no reducible proper subterms may be reduced. More precisely, the *innermost rewrite relation* $\to^i_{\mathcal{R}}$ induced by $\mathcal{R}$ is defined as follows:

(i) $\ell\gamma \to^i_{\mathcal{R}} r\gamma$ if $\ell \to r \in \mathcal{R}$ and all proper subterms of $\ell\gamma$ are in normal form,

(ii) $s\ t \to^i_{\mathcal{R}} s'\ t$ if $s \to^i_{\mathcal{R}} s'$, and

(iii) $s\ t \to^i_{\mathcal{R}} s\ t'$ if $t \to^i_{\mathcal{R}} t'$.

Below we only analyze innermost rewriting. So we write $\to$ for $\to^i_{\mathcal{R}}$ whenever no ambiguity arises.

**Derivation Height and Complexity.** Given a relation $\to$ on terms, we write $s \xrightarrow{n} t$ if there is a sequence $s = s_0 \to \cdots \to s_n = t$ of length $n$. The *derivation height* $\mathtt{dh}(s,\to)$ of a term $s$ with respect to $\to$ is the length of the longest $\to$-sequence of starting with $s$, i.e., $\mathtt{dh}(s,\to) = \max\{n \mid \exists t \in T(\mathcal{F},\mathcal{X}) : s \xrightarrow{n} t\}$. The *absolute size* of a term $s$, denoted by $|s|$, is 1 if $s$ is a symbol in $\mathcal{F}$ or a variable, and $|s_1| + |s_2|$ if $s = s_1\ s_2$. In order to express various complexity notions in the rewriting setting, we define the *complexity function* as follows: $\mathtt{comp}(n,\to,\mathcal{T}) = \max\{\mathtt{dh}(s,\to) \mid s \in \mathcal{T} \text{ and } |s| \le n\}$. Intuitively, $\mathtt{comp}(n,\to,\mathcal{T})$ is the length of the longest $\to$-sequence starting with a term whose absolute size is at most $n$ from $\mathcal{T}$. We summarize four particular instances in the following table:

| | derivational | runtime |
|---|---|---|
| full | $\mathtt{dc}_{\mathcal{R}}(n) = \mathtt{comp}(n,\to_{\mathcal{R}},T(\mathcal{F},\mathcal{X}))$ | $\mathtt{rc}_{\mathcal{R}}(n) = \mathtt{comp}(n,\to_{\mathcal{R}},T_b(\mathcal{F}))$ |
| innermost | $\mathtt{idc}_{\mathcal{R}}(n) = \mathtt{comp}(n,\to^i_{\mathcal{R}},T(\mathcal{F},\mathcal{X}))$ | $\mathtt{irc}_{\mathcal{R}}(n) = \mathtt{comp}(n,\to^i_{\mathcal{R}},T_b(\mathcal{F}))$ |

**Ordered Sets and Monotonic Functions.** A *quasi-ordered set* $(A,\sqsupseteq)$ consists of a nonempty set $A$ and a quasi-order (reflexive and transitive) $\sqsupseteq$ on $A$. An *extended well-founded set* $(A,>,\ge)$ is a nonempty set $A$ together with a well-founded order $>$ and a quasi-order $\ge$ on $A$ such that $\ge$ is compatible with $>$, i.e., $x > y$ implies $x \ge y$ and $x > y \ge z$ implies $x > z$. Below we refer to an extended well-founded set simply as a *well-founded set*.

Given quasi-ordered sets $(A,\sqsupseteq)$ and $(B,\sqsupseteq)$, a function $f : A \longrightarrow B$ is said to be *weakly monotonic* if $x \sqsupseteq y$ implies $f(x) \sqsupseteq f(y)$. Let $A \Longrightarrow B$ denote the set of weakly monotonic functions from $A$ to $B$. The comparison operator $\sqsupseteq$ on $B$ induces pointwise comparison on $A \Longrightarrow B$ as follows: $f \sqsupseteq g$ if $f(x) \sqsupseteq g(x)$ for all $x \in A$. This way $(A \Longrightarrow B,\sqsupseteq)$ is also a quasi-ordered set. Given well-founded sets $(A,>,\ge)$ and $(B,>,\ge)$, a function $f : A \longrightarrow B$ is said to be *strongly monotonic* if $x > y$ implies $f(x) > f(y)$ and $x \ge y$ implies $f(x) \ge f(y)$.

# 3 Tuple Interpretations

In this section, we introduce the notion of tuple algebras in the context of innermost rewriting. We start by interpreting types as cost–size products, give interpretation of terms as cost–size tuples, and finally prove the innermost version of the compatibility theorem.

## 3.1 Types as Cost–Size Products

We start with defining cost–size products.

**Definition 1 (Cost–Size Products)** Given a well-founded set $(\mathcal{C}, >, \geq)$, called the *cost set*, and a quasi-ordered set $(\mathcal{S}, \sqsupseteq)$, called the *size set*, we call $\mathcal{C} \times \mathcal{S}$ the *cost–size product* of $(\mathcal{C}, >, \geq)$ and $(\mathcal{S}, \sqsupseteq)$, and its elements *cost–size tuples*.

Cost–size tuples can be ordered as follows:

**Definition 2 (Product Order)** For all $\langle x, y\rangle$ and $\langle x', y'\rangle$ in $\mathcal{C} \times \mathcal{S}$,

(i) $\langle x, y\rangle \succ \langle x', y'\rangle$ if $x > x'$ and $y \sqsupseteq y'$, and

(ii) $\langle x, y\rangle \succcurlyeq \langle x', y'\rangle$ if $x \geq x'$ and $y \sqsupseteq y'$.

And we get a well-founded set.

**Lemma 1** *$(\mathcal{C} \times \mathcal{S}, \succ, \succcurlyeq)$ is a well-founded set.*

PROOF It follows immediately from the definition that $\succ$ and $\succcurlyeq$ are transitive, and $\succcurlyeq$ is reflexive. To prove that $\succ$ is well-founded, note that the existence of $\langle x_1, y_1\rangle \succ \langle x_2, y_2\rangle \succ \cdots$ would imply $x_1 > x_2 > \cdots$, which cannot be the case since $>$ is well-founded.

We still need to check that $\succcurlyeq$ is compatible with $\succ$.

- Suppose $\langle x, y\rangle \succ \langle x', y'\rangle$. Since $x > x'$ implies $x \geq x'$, we have $\langle x, y\rangle \succcurlyeq \langle x', y'\rangle$.
- Suppose $\langle x, y\rangle \succ \langle x', y'\rangle \succcurlyeq \langle x'', y''\rangle$. Since $x > x' \geq x''$ implies $x > x''$ and $\sqsupseteq$ is transitive, we have $\langle x, y\rangle \succ \langle x'', y''\rangle$. ∎

Now we interpret types as a particular kind of cost–size products.

**Definition 3 (Interpretation of Types)** Let $\mathcal{B}$ denote the set of sorts. An *interpretation key* $\mathcal{J}_\mathcal{B}$ for $\mathcal{B}$ maps each sort $\iota$ to a quasi-ordered set $(\mathcal{J}_\mathcal{B}(\iota), \sqsupseteq)$ with a minimum. For each type $\sigma \in \mathcal{T}_\mathcal{B}$, we define the cost–size interpretation of $\sigma$ as the product $(\!|\sigma|\!) = \mathcal{C}_\sigma \times \mathcal{S}_\sigma$ with

$$\begin{aligned} \mathcal{C}_\sigma &= \mathbb{N} \times \mathcal{F}^{\mathsf{c}}_\sigma \\ \mathcal{F}^{\mathsf{c}}_\iota &= \mathtt{unit} & \mathcal{S}_\iota &= \mathcal{J}_\mathcal{B}(\iota) \\ \mathcal{F}^{\mathsf{c}}_{\iota\Rightarrow\tau} &= \mathcal{S}_\iota \Longrightarrow \mathcal{C}_\tau & \mathcal{S}_{\iota\Rightarrow\tau} &= \mathcal{S}_\iota \Longrightarrow \mathcal{S}_\tau \end{aligned}$$

where $\mathtt{unit} = \{\mathfrak{U}\}$ is quasi-ordered by $\geq$ with $\mathfrak{U} \geq \mathfrak{U}$. All $\mathcal{F}^{\mathsf{c}}_{\iota\Rightarrow\tau}$ and $\mathcal{S}_{\iota\Rightarrow\tau}$ are ordered by pointwise comparison. The set $\mathcal{C}_\sigma$ is ordered as follows: $(n, f) > (m, g)$ if $n > m$ and $f \geq g$, and $(n, f) \geq (m, g)$ if $n \geq m$ and $f \geq g$. This definition requires that all $(\mathcal{C}_\sigma, \geq)$ and $(\mathcal{S}_\sigma, \sqsupseteq)$ are quasi-ordered sets, which is guaranteed by the following lemma.

**Lemma 2** *For any type $\sigma$, $(\mathcal{C}_\sigma, >, \geq)$ is a well-founded set and $(\mathcal{S}_\sigma, \sqsupseteq)$ is a quasi-ordered set with a minimum. Therefore, $(\!|\sigma|\!)$ is a cost–size product.*

PROOF When $\sigma$ is a sort, $\mathcal{C}_\sigma = \mathbb{N} \times \mathtt{unit} \cong \mathbb{N}$ and $\mathcal{S}_\sigma = \mathcal{J}_\mathcal{B}(\sigma)$, so the statement is trivially true. When $\sigma = \iota \Rightarrow \tau$, we have $\mathcal{C}_\sigma = \mathbb{N} \times \mathcal{F}^{\mathsf{c}}_{\iota\Rightarrow\tau}$, $\mathcal{F}^{\mathsf{c}}_{\iota\Rightarrow\tau} = \mathcal{J}_\mathcal{B}(\iota) \Longrightarrow \mathcal{C}_\tau$ and $\mathcal{S}_\sigma = \mathcal{J}_\mathcal{B}(\iota) \Longrightarrow \mathcal{S}_\tau$. By induction, $(\mathcal{C}_\tau, \geq)$ and $(\mathcal{S}_\tau, \sqsupseteq)$ are quasi-ordered sets. So are $(\mathcal{F}^{\mathsf{c}}_{\iota\Rightarrow\tau}, \geq)$ and $(\mathcal{S}_\sigma, \sqsupseteq)$, which are ordered by pointwise comparison. By Lemma 1, $(\mathcal{C}_\sigma, >, \geq)$ is a well-founded set. One minimum of $(\mathcal{S}_\sigma, \sqsupseteq)$ is the constant function $\boldsymbol{\lambda} x.\bot$ where $\bot$ is a minimum of $(\mathcal{S}_\tau, \sqsupseteq)$. ∎

The cost component $\mathcal{C}_\sigma$ of $(\!|\sigma|\!)$ holds information about the cost of reducing a term of type $\sigma$ to its normal form. It has two parts: one is numeric and the other is functional. The functional part $\mathcal{F}^{\mathsf{c}}_\sigma$ degenerates to $\mathtt{unit}$ when $\sigma$ is just a sort, and is indeed a function space when $\sigma = \iota \Rightarrow \tau$ is a function type. In the latter case, $\mathcal{F}^{\mathsf{c}}_\sigma = \mathcal{S}_\iota \Longrightarrow \mathcal{C}_\tau$ consists of functions with domain $\mathcal{S}_\iota$, the size component of $(\!|\iota|\!)$. This is very much in line with the standard complexity notion based on Turing machines, where time complexity is parametrized by the size of input.

In order to use Definition 3 to interpret types, we need a concrete interpretation key, which chooses a size set for each sort. In our examples, a particular kind of interpretation keys map each sort $\iota$ to $(\mathbb{N}^{K[\iota]}, \sqsupseteq)$ where $K[\iota] \geq 1$ and $\langle x_1, \ldots, x_{K[\iota]} \rangle \sqsupseteq \langle y_1, \ldots, y_{K[\iota]} \rangle$ if $x_i \geq y_i$ for all $i$. Such interpretation keys are used unless otherwise stated. We take a semantic approach (cf. [16]) to determine the number $K[\iota]$ for each sort $\iota$. For example, $\mathsf{nat}$ is the sort of natural numbers in unary and $n \in \mathbb{N}$ is represented as $\mathsf{s}(\ldots(\mathsf{s}\ \mathsf{0}))$ with $n$ successive applications of $\mathsf{s}$. The number of occurrences of $\mathsf{s}$ is a reasonable measure for the size of a natural number so we let $K[\mathsf{nat}]$ be 1. On the other hand, to characterize the size of a list, we need information about the individual elements in addition to the length of the list. So for each list, we keep track of its length as well as the maximum size of its elements. This way $K[\mathsf{list}] = 2$. See Example 2.

**Definition 4** Cost–size tuples in $(\!|\sigma|\!)$ are written as $\langle (n, f^{\mathsf{c}}), f^{\mathsf{s}} \rangle$ where $n \in \mathbb{N}$, $f^{\mathsf{c}} \in \mathcal{F}^{\mathsf{c}}_\sigma$ and $f^{\mathsf{s}} \in \mathcal{S}_\sigma$. When $\sigma$ is a function type, we refer to $f^{\mathsf{c}}$ as the *cost function* and $f^{\mathsf{s}}$ as the *size function*.

In order to define the interpretation of terms (Definition 7), we need a notion of application for cost–size tuples. Given $\boldsymbol{f} \in (\!|\iota \Rightarrow \tau|\!)$ and $\boldsymbol{x} \in (\!|\iota|\!)$, our goal is to define $\boldsymbol{f} \cdot \boldsymbol{x} \in (\!|\tau|\!)$. Let us demonstrate with an example. Recall from Example 1 the function $\mathtt{append} : \mathsf{list} \Rightarrow \mathsf{list} \Rightarrow \mathsf{list}$, which takes two lists $q$ and $l$ as input. Let $\mathtt{append}$ be interpreted as $\boldsymbol{f} = \langle (n, f^{\mathsf{c}}), f^{\mathsf{s}} \rangle \in (\!|\mathsf{list} \Rightarrow \mathsf{list} \Rightarrow \mathsf{list}|\!)$, where

$$
\begin{aligned}
& n \in \mathbb{N}, \\
& f^{\mathsf{c}} \in \overbrace{\mathcal{S}_{\mathsf{list}}}^{\text{size of } q} \Longrightarrow (\mathbb{N} \times (\overbrace{\mathcal{S}_{\mathsf{list}}}^{\text{size of } l} \Longrightarrow (\mathbb{N} \times \mathtt{unit}))), \text{ and} \\
& f^{\mathsf{s}} \in \overbrace{\mathcal{S}_{\mathsf{list}}}^{\text{size of } q} \Longrightarrow (\overbrace{\mathcal{S}_{\mathsf{list}}}^{\text{size of } l} \Longrightarrow \mathcal{S}_{\mathsf{list}}).
\end{aligned}
$$

For the first list $q$, take a cost–size tuple $\boldsymbol{x} = \langle (m, \mathfrak{U}), x^{\mathsf{s}} \rangle$ from $(\!|\mathsf{list}|\!)$. We apply $f^{\mathsf{c}}$ and $f^{\mathsf{s}}$ to $x^{\mathsf{s}}$, and get $f^{\mathsf{c}}(x^{\mathsf{s}}) = (k, h) \in \mathbb{N} \times (\mathcal{S}_{\mathsf{list}} \Longrightarrow (\mathbb{N} \times \mathtt{unit}))$ and $f^{\mathsf{s}}(x^{\mathsf{s}}) \in \mathcal{S}_{\mathsf{list}} \Longrightarrow \mathcal{S}_{\mathsf{list}}$, respectively. Then we sum the numeric parts and collect all the data in the new cost–size tuple $\langle (n + m + k, h), f^{\mathsf{s}}(x^{\mathsf{s}}) \rangle$. This process is summarized in the following definition.

**Definition 5 (Semantic Application)** Given $\boldsymbol{f} = \langle (n, f^{\mathsf{c}}), f^{\mathsf{s}} \rangle \in (\!|\iota \Rightarrow \tau|\!)$ and $\boldsymbol{x} = \langle (m, \mathfrak{U}), x^{\mathsf{s}} \rangle \in (\!|\iota|\!)$, the *semantic application* of $\boldsymbol{f}$ to $\boldsymbol{x}$, denoted by $\boldsymbol{f} \cdot \boldsymbol{x}$, is $\langle (n + m + k, h), f^{\mathsf{s}}(x^{\mathsf{s}}) \rangle$ where $f^{\mathsf{c}}(x^{\mathsf{s}}) = (k, h)$.

Semantic application is left-associative, so $\boldsymbol{f} \cdot \boldsymbol{g} \cdot \boldsymbol{h}$ stands for $(\boldsymbol{f} \cdot \boldsymbol{g}) \cdot \boldsymbol{h}$. This definition conforms to the types, which is stated in the following lemma.

**Lemma 3** *If $\boldsymbol{f} \in (\!|\iota \Rightarrow \tau|\!)$ and $\boldsymbol{x} \in (\!|\iota|\!)$, then $\boldsymbol{f} \cdot \boldsymbol{x} \in (\!|\tau|\!)$.*

**Remark 1** Because $\mathbb{N} \times \mathtt{unit}$ is order-isomorphic to $\mathbb{N}$, we identify $\mathbb{N} \times \mathtt{unit}$ with $\mathbb{N}$ and $(m, \mathfrak{U})$ with $m$ unless otherwise stated. So we write $\langle m, x^{\mathsf{s}} \rangle$ for cost–size tuples in $(\!|\iota|\!)$ where $\iota$ is a sort.

### 3.2 Cost–Size Tuple Algebras

**Definition 6** A *cost–size tuple algebra* $(\langle\!|\cdot|\!\rangle, \mathcal{J})$ over a syntax signature $(\mathcal{B}, \mathcal{F}, \mathtt{ar})$ consists of

(i) a family of cost–size products $\{\langle\!|\sigma|\!\rangle\}_{\sigma \in \mathcal{T}_\mathcal{B}}$, and

(ii) an interpretation function $\mathcal{J} : \mathcal{F} \longrightarrow \biguplus_\sigma \langle\!|\sigma|\!\rangle$ which associates to each $\mathsf{f} : \sigma$ an element $\mathcal{J}_\mathsf{f} \in \langle\!|\sigma|\!\rangle$.

With innermost rewriting, we assume that variables have no cost.

**Definition 7** Fix a cost–size tuple algebra $(\langle\!|\cdot|\!\rangle, \mathcal{J})$. A *valuation* $\alpha : \mathcal{X} \longrightarrow \biguplus_\sigma \langle\!|\sigma|\!\rangle$ is a function which maps each variable $x : \iota$ to a zero-cost tuple $\langle 0, x^\mathsf{s} \rangle \in \langle\!|\iota|\!\rangle$. The interpretation of a term $s$ under valuation $\alpha$, denoted by $[\![s]\!]^\mathcal{J}_\alpha$, is defined as follows:

$$[\![\mathsf{f}]\!]^\mathcal{J}_\alpha = \mathcal{J}_\mathsf{f} \qquad [\![x]\!]^\mathcal{J}_\alpha = \alpha(x) \qquad [\![s\ t]\!]^\mathcal{J}_\alpha = [\![s]\!]^\mathcal{J}_\alpha \cdot [\![t]\!]^\mathcal{J}_\alpha$$

As a corollary of Lemma 3, interpretation of terms conforms with types.

**Lemma 4** *If $s : \sigma$ then $[\![s]\!]^\mathcal{J}_\alpha$ is in $\langle\!|\sigma|\!\rangle$, for all valuations $\alpha$.*

Let $\sigma$ be $\iota_1 \Rightarrow \ldots \Rightarrow \iota_m \Rightarrow \kappa$ where all $\iota_i$ and $\kappa$ are sorts. Elements of $\mathcal{C}_\sigma$ can be written as

$$\begin{array}{l} (e_0, \boldsymbol{\lambda} x_1. \\ \qquad (e_1, \boldsymbol{\lambda} x_2. \\ \qquad\qquad \ldots \\ \qquad\qquad (e_{m-1}, \boldsymbol{\lambda} x_m. \\ \qquad\qquad\qquad (e_m, \mathfrak{U})) \ldots)). \end{array} \tag{1}$$

When $e_0 = e_1 = \cdots = e_{m-1} = 0$, we write $(\boldsymbol{\lambda} x_1 \ldots x_m . e_m)$ as a shorthand.

**Example 2** Let $\mathcal{S}_\mathsf{nat}$ and $\mathcal{S}_\mathsf{list}$ be $\mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$, respectively. Recall that the size of a natural number is the number of occurrences of $\mathsf{s}$, and the size of a list is a pair $q = (q_\mathsf{l}, q_\mathsf{m})$ where $q_\mathsf{l}$ is the length and $q_\mathsf{m}$ is the maximum size of the elements. We interpret the constructors as follows:

$$\begin{array}{ll} \mathcal{J}_\mathsf{0} = \langle 0, 0 \rangle & \mathcal{J}_\mathsf{s} = \langle (\boldsymbol{\lambda} x.0), \boldsymbol{\lambda} x.x + 1 \rangle \\ \mathcal{J}_\mathsf{nil} = \langle 0, (0,0) \rangle & \mathcal{J}_\mathsf{cons} = \langle (\boldsymbol{\lambda} xq.0), \boldsymbol{\lambda} xq.(q_\mathsf{l} + 1, \max(x, q_\mathsf{m})) \rangle \end{array}$$

Both $\mathsf{0}$ and $\mathsf{nil}$ have no cost because they are constructors without a function type. With innermost rewriting, constructors with a function type, such as $\mathsf{s}$ and $\mathsf{cons}$, have $e_0 = \cdots = e_m = 0$ for cost of form (1).

### 3.3 Compatibility Theorem

Roughly, the compatibility theorem (Theorem 1) states that if $\mathcal{R}$ is compatible with a tuple algebra $\mathcal{A}$, then the rewriting relation $\to^i_\mathcal{R}$ is embedded in the well-founded order on cost–size products. The next two lemmas are technical results needed in order to prove it. Lemma 5 states that interpretations are closed under substitution and Lemma 6 provides strong monotonicity to semantic application.

**Definition 8** Fix a cost–size tuple algebra $(\langle\!|\cdot|\!\rangle, \mathcal{J})$. A substitution $\gamma$ is *zero-cost* under valuation $\alpha$ if $[\![\gamma(x)]\!]^\mathcal{J}_\alpha$ is a zero-cost tuple for each variable $x$. Given a valuation $\alpha$ and a zero-cost substitution $\gamma$, the function $\alpha^\gamma = [\![\cdot]\!]^\mathcal{J}_\alpha \circ \gamma = [\![\gamma(\cdot)]\!]^\mathcal{J}_\alpha$ is thus a valuation.

**Lemma 5 (Substitution)** *If $\gamma$ is a zero-cost substitution under valuation $\alpha$, $[\![s\gamma]\!]^{\mathcal{J}}_{\alpha} = [\![s]\!]^{\mathcal{J}}_{\alpha^\gamma}$ for any term $s$.*

**Lemma 6** $\mathtt{App}(\boldsymbol{f},\boldsymbol{x}) = \boldsymbol{f}\cdot\boldsymbol{x}$ *is strongly monotonic on both arguments.*

PROOF We need to prove (i) if $\boldsymbol{f} \succ \boldsymbol{g}$ and $\boldsymbol{x} \succcurlyeq \boldsymbol{y}$, then $\mathtt{App}(\boldsymbol{f},\boldsymbol{x}) \succ \mathtt{App}(\boldsymbol{g},\boldsymbol{y})$; (ii) if $\boldsymbol{f} \succcurlyeq \boldsymbol{g}$ and $\boldsymbol{x} \succ \boldsymbol{y}$, then $\mathtt{App}(\boldsymbol{f},\boldsymbol{x}) \succ \mathtt{App}(\boldsymbol{g},\boldsymbol{y})$; (iii) if $\boldsymbol{f} \succcurlyeq \boldsymbol{g}$ and $\boldsymbol{x} \succcurlyeq \boldsymbol{y}$, then $\mathtt{App}(\boldsymbol{f},\boldsymbol{x}) \succcurlyeq \mathtt{App}(\boldsymbol{g},\boldsymbol{y})$. Consider cost–size tuples $\boldsymbol{f},\boldsymbol{g} \in (\!|\iota \Rightarrow \tau|\!)$ and $\boldsymbol{x},\boldsymbol{y} \in (\!|\iota|\!)$. Let $\boldsymbol{f} = \langle (n, f^{\mathsf{c}}), f^{\mathsf{s}}\rangle$, $\boldsymbol{g} = \langle (m, g^{\mathsf{c}}), g^{\mathsf{s}}\rangle$, $\boldsymbol{x} = \langle x^{\mathsf{c}}, x^{\mathsf{s}}\rangle$, and $\boldsymbol{y} = \langle y^{\mathsf{c}}, y^{\mathsf{s}}\rangle$. We proceed to show (i) and observe that (ii) and (iii) follow similar reasoning. Indeed, if $\boldsymbol{f} \succ \boldsymbol{g}$ and $\boldsymbol{x} \succcurlyeq \boldsymbol{y}$ we have that $n > m$, $f^{\mathsf{c}} \geq g^{\mathsf{c}}$, $f^{\mathsf{s}} \sqsupseteq g^{\mathsf{s}}$, $x^{\mathsf{c}} \geq y^{\mathsf{c}}$, and $x^{\mathsf{s}} \sqsupseteq y^{\mathsf{s}}$. Hence, by letting $f^{\mathsf{c}}(x^{\mathsf{s}}) = (k,h)$ and $g^{\mathsf{c}}(y^{\mathsf{s}}) = (k',h')$, we get:

$$\mathtt{App}(\boldsymbol{f},\boldsymbol{x}) = \langle (n, f^{\mathsf{c}}), f^{\mathsf{s}}\rangle \cdot \langle x^{\mathsf{c}}, x^{\mathsf{s}}\rangle = \langle (n + x^{\mathsf{c}} + k, h), f^{\mathsf{s}}(x^{\mathsf{s}})\rangle > \left\langle (m + y^{\mathsf{c}} + k', h'), g^{\mathsf{s}}(y^{\mathsf{s}})\right\rangle = \mathtt{App}(\boldsymbol{g},\boldsymbol{y})$$

■

**Definition 9** A TRS $\mathcal{R}$ is said to be *compatible* with a cost–size tuple algebra $((\!|\cdot|\!),\mathcal{J})$ if $[\![\ell]\!]^{\mathcal{J}}_{\alpha} \succ [\![r]\!]^{\mathcal{J}}_{\alpha}$ for all rules $\ell \to r \in \mathcal{R}$ and valuations $\alpha$.

**Theorem 1 (Compatibility)** *Let $\mathcal{R}$ be a TRS compatible with a cost–size tuple algebra $((\!|\cdot|\!),\mathcal{J})$. Then, for any pair of terms $s$ and $t$, whenever $s \to^{i}_{\mathcal{R}} t$ we have $[\![s]\!]^{\mathcal{J}}_{\alpha} \succ [\![t]\!]^{\mathcal{J}}_{\alpha}$.*

PROOF We proceed by induction on $\to^{i}_{\mathcal{R}}$. For the base case, $s \to^{i}_{\mathcal{R}} t$ by $\ell\gamma \to r\gamma$ and all subterms of $\ell\gamma$ are in $\to_{\mathcal{R}}$ normal form. Therefore, since $[\![\ell]\!]^{\mathcal{J}}_{\alpha} \succ [\![r]\!]^{\mathcal{J}}_{\alpha}$ by hypothesis, Lemma 5 gives us that $[\![\ell\gamma]\!]^{\mathcal{J}}_{\alpha} \succ [\![r\gamma]\!]^{\mathcal{J}}_{\alpha}$.

In the inductive step we use Lemma 6 combined with the (IH) as follows. Suppose $s \to^{i}_{\mathcal{R}} t$ by $s = s'u \to^{i}_{\mathcal{R}} s''u$ with $s' \to^{i}_{\mathcal{R}} s''$. Hence, $[\![s'u]\!]^{\mathcal{J}}_{\alpha} = [\![s']\!]^{\mathcal{J}}_{\alpha} \cdot [\![u]\!]^{\mathcal{J}}_{\alpha} = \mathtt{App}([\![s']\!]^{\mathcal{J}}_{\alpha}, [\![u]\!]^{\mathcal{J}}_{\alpha})$, henceforth the induction hypothesis gives $[\![s']\!]^{\mathcal{J}}_{\alpha} \succ [\![s'']\!]^{\mathcal{J}}_{\alpha}$, which combined with Lemma 6 implies $[\![s]\!]^{\mathcal{J}}_{\alpha} = \mathtt{App}([\![s']\!]^{\mathcal{J}}_{\alpha}, [\![u]\!]^{\mathcal{J}}_{\alpha}) \succ \mathtt{App}([\![s'']\!]^{\mathcal{J}}_{\alpha}, [\![u]\!]^{\mathcal{J}}_{\alpha}) = [\![t]\!]^{\mathcal{J}}_{\alpha}$. When $s \to^{i}_{\mathcal{R}} t$ with $s = s'\ u \to^{i}_{\mathcal{R}} s'\ u'$ the proof is analogous. ■

**Example 3** Let $\mathsf{0},\mathsf{1} : \iota$, $\mathsf{g} : \iota \Rightarrow \iota \Rightarrow \iota$, and $\mathsf{f} : \iota \Rightarrow \iota \Rightarrow \iota \Rightarrow \iota$. The rewrite system introduced by Toyama [22] and defined by $\mathcal{R} = \{\mathsf{g}\ x\ y \to x, \mathsf{g}\ x\ y \to y, \mathsf{f}\ \mathsf{0}\ \mathsf{1}\ z \to \mathsf{f}\ z\ z\ z\}$ was given to show that termination is not modular for disjoint unions of TRSs. Indeed, it admits the infinite rewriting sequence $\mathsf{f}\ \mathsf{0}\ \mathsf{1}\ (\mathsf{g}\ \mathsf{0}\ \mathsf{1}) \to_{\mathcal{R}} \mathsf{f}\ (\mathsf{g}\ \mathsf{0}\ \mathsf{1})\ (\mathsf{g}\ \mathsf{0}\ \mathsf{1})\ (\mathsf{g}\ \mathsf{0}\ \mathsf{1}) \to^{+}_{\mathcal{R}} \mathsf{f}\ \mathsf{0}\ \mathsf{1}\ (\mathsf{g}\ \mathsf{0}\ \mathsf{1})$. However, the innermost relation $\to^{i}_{\mathcal{R}}$ is terminating. In order to prove it, we introduce a non-numeric notion of size. Let $\mathcal{J}_{\mathcal{B}}(\iota) = \mathcal{P}(T(\mathcal{F},\mathcal{X}))$, i.e., the set of all subsets of $T(\mathcal{F},\mathcal{X})$. This set is partially ordered by set inclusion, so $x \sqsupseteq y$ iff $x \supseteq y$, which is a quasi-order. Consider the following interpretation:

$$\mathcal{J}_0 = \langle 0, \{0\}\rangle \qquad \mathcal{J}_1 = \langle 0, \{1\}\rangle \qquad \mathcal{J}_{\mathsf{g}} = \langle (\boldsymbol{\lambda} xy.1), \boldsymbol{\lambda} xy.x \cup y\rangle \qquad \mathcal{J}_{\mathsf{f}} = \langle (\boldsymbol{\lambda} xyz.H(x,y)), \boldsymbol{\lambda} xyz.\emptyset\rangle\,,$$

where $H$ is a helper function defined by $H(x,y) = \mathtt{if}\ x \sqsupseteq \{0\} \wedge y \sqsupseteq \{1\}\ \mathtt{then}\ 1\ \mathtt{else}\ 0$. Notice that $H$ is weakly monotonic and all terms in normal form are interpreted as sets of size $\leq 1$. Checking compatibility is straightforward: $[\![\mathsf{g}\ x\ y]\!] = \langle 1, x \cup y\rangle \succ \langle 0, x\rangle = [\![x]\!]$ and $[\![\mathsf{g}\ x\ y]\!] = \langle 1, x \cup y\rangle \succ \langle 0, y\rangle = [\![y]\!]$; and $[\![\mathsf{f}\ \mathsf{0}\ \mathsf{1}\ z]\!] = \langle 1, \emptyset\rangle \succ \langle 0, \emptyset\rangle = [\![\mathsf{f}\ z\ z\ z]\!]$, because any instantiation of $z$ is necessarily in normal form, so it cannot include both $\mathsf{0}$ and $\mathsf{1}$.

This example, albeit artificial, is interesting from a termination point of view. It shows that tuple interpretations can be used to deal with rewrite systems that only terminate via the innermost strategy.

# 4 Polynomial Bounds for Innermost Runtime Complexity

In this section, we study the applications of tuple interpretations to complexity analysis of compatible TRSs, i.e., rewriting systems that admit an interpretation in a tuple algebra $((\!|\cdot|\!),\mathcal{J})$. Even though cost and size are split in our setting, they are intertwined concepts (in a sense we make precise in this section) that constitute what we intuitively call "complexity" of a TRS.

### 4.1 Additive Tuple Interpretations

In order to establish upper bounds to $\mathtt{irc}_{\mathcal{R}}(n)$, it suffices to bound the cost component $[\![s]\!]^{\mathsf{c}}$ of all terms $s$ where $|s| \leq n$. Furthermore, since basic terms are of the form $\mathsf{f}\ d_1 \dots d_m$, the size of data terms plays an important role in our analysis. In what follows, we use the default choice for interpretation key when interpreting types; that is, $\mathcal{J}_{\mathcal{B}}(\iota) = \mathbb{N}^{K[\iota]}$, with $K[\iota] \geq 1$ for each $\iota \in \mathcal{B}$.

Given $\sigma = \iota_1 \Rightarrow \ldots \Rightarrow \iota_m \Rightarrow \kappa$, the size component of $(\!|\sigma|\!)$ is $\mathcal{S}_\sigma = \mathbb{N}^{K[\iota_1]} \Longrightarrow \ldots \Longrightarrow \mathbb{N}^{K[\iota_m]} \Longrightarrow \mathbb{N}^{K[\kappa]}$. Size functions $f^{\mathsf{s}} \in \mathcal{S}_\sigma$ when fully applied can be written in terms of functional components. Hence, $f^{\mathsf{s}}(x_1,\dots,x_m) = \left\langle f^{\mathsf{s}}_1(x_1,\dots,x_m),\dots,f^{\mathsf{s}}_{K[\kappa]}(x_1,\dots,x_m)\right\rangle$.

**Definition 10** Let $\sigma$ be a type and $f^{\mathsf{s}} \in \mathcal{S}_\sigma$. The size function $f^{\mathsf{s}}$ is *linearly bounded* if each one of its component functions $f^{\mathsf{s}}_1,\dots,f^{\mathsf{s}}_{K[\kappa]}$ is upper-bounded by a positive linear polynomial, i.e., there is a positive constant $a \in \mathbb{N}$ such that for all $1 \leq l \leq m$, $f^{\mathsf{s}}_l(x_1,\dots,x_m) \leq a(1+\sum_{i=1}^{m}\sum_{j=1}^{K[\iota_i]} x_{ij})$. Analogously, we say $f^{\mathsf{s}}$ is *additive* if there is a constant $a \in \mathbb{N}$ such that $\sum_{l=1}^{K[\kappa]} f^{\mathsf{s}}_l(x_1,\dots,x_m) \leq a+\sum_{i=1}^{m}\sum_{j=1}^{K[\iota_i]} x_{ij}$.

Notice that by this definition linearly bounded (or additive) size functions are not required to be linear (or additive) but to be upper-bounded by a linear (additive) function. So this permits us to use for instance $\min(x,2y)$, whereas $xy$ cannot be used. Size interpretations do not necessarily bound the absolute size of data terms. For instance, we may interpret a data constructor $\mathsf{c}\colon \iota \Rightarrow \kappa$ with $\mathcal{J}^{\mathsf{s}}_{\mathsf{c}} = \boldsymbol{\lambda} x.\lfloor x/2 \rfloor$ which would give us $|d| \geq [\![d]\!]^s$. This is specially useful when dealing with sublinear interpretations.

The next lemma ensures that by interpreting constructors additively the size interpretation of data terms is proportional to their absolute size:

**Lemma 7** *Let $\mathcal{R}$ be a TRS compatible with a cost–size tuple algebra $(\!|\cdot|\!),\mathcal{J})$.*

*(i) Assume $\mathcal{J}^{\mathsf{s}}_{\mathsf{c}}$ is additive for all data constructors $\mathsf{c}$, then for all data terms $d$: if $|d| \leq n$, then there exists a constant $b > 0$ such that $[\![d]\!]^{\mathsf{s}}_l \leq bn$, for each size-component $[\![d]\!]^{\mathsf{s}}_l$ of $[\![d]\!]$.*

*(ii) Assume $\mathcal{J}^{\mathsf{s}}_{\mathsf{c}}$ is linearly bounded for all data constructors $\mathsf{c}$, then for all data terms $d$: if $|d| \leq n$, then there exists a constant $b > 0$ such that $[\![d]\!]^{\mathsf{s}}_l \leq 2^{bn}$, for each size-component $[\![d]\!]^{\mathsf{s}}_l$ of $[\![d]\!]$.*

The bound in (ii) is sharp. Indeed, define (when interpreting $\mathcal{R}_{\mathtt{add}}$): $\mathcal{J}_0 = \langle 0,1\rangle$, $\mathcal{J}_{\mathsf{s}} = \langle(\boldsymbol{\lambda} x.0),\boldsymbol{\lambda} x.2x+1\rangle$, and $\mathcal{J}_{\mathtt{add}} = \langle(\boldsymbol{\lambda} xy.y+1),\boldsymbol{\lambda} xy.x+y\rangle$. In this case, for a data term $\mathsf{n} = \mathsf{s}^n(0)$ its size interpretation is exactly $[\![\mathsf{n}]\!]^{\mathsf{s}} = 2^n + n \leq 2^{|\mathsf{n}|}$. However, whereas this choice is compatible with $\mathcal{R}_{\mathtt{add}}$, and hence proving its termination, it induces an exponential overhead on $\mathtt{irc}_{\mathcal{R}_{\mathtt{add}}}$, which is linearly bounded (see Example 4). Such a huge overestimation is not desirable in a complexity analysis setting. This behavior sets a strict upper-bound to the interpretation of data constructors; namely, we seek to bound constructor's size interpretations additively. It is easy to show that size components for nat and list in Example 2 are additive.

**Definition 11** We say an interpretation $\mathcal{J}$ is additive if for each $\mathsf{c} \in \mathcal{C}$, $\mathcal{J}^{\mathsf{s}}_{\mathsf{c}}$ is additive.

### 4.2 Cost-Bounded Tuple Interpretations

In what follows, we consider rewriting systems with additive interpretations.

**Definition 12** Let $\sigma$ be a type and $\boldsymbol{f}^{\mathsf{c}} \in \mathcal{C}_\sigma$. We say $\boldsymbol{f}^{\mathsf{c}}$, written as in form (1), is linearly (additively) bounded whenever each $e_i$, $0 \leq i \leq m$, is linearly (additively) bounded. Additionally, $\mathcal{J}_{\mathsf{f}}$ is bounded by a functional $f$ if both $\mathcal{J}^{\mathsf{c}}_{\mathsf{f}}$ and $\mathcal{J}^{\mathsf{s}}_{\mathsf{f}}$ are bounded by $f$.

In the next lemma, we collect the appropriate induced upper-bounds on innermost runtime complexity given that we can provide bounds to the cost–size components of interpretations.

**Lemma 8** *Suppose $\mathcal{R}$ is a TRS compatible with a tuple algebra $(\!|\cdot|\!), \mathcal{J})$, then:*

*(i) if, for all $\mathsf{f} \in \mathcal{F}$, $\mathcal{J}_\mathsf{f}^\mathsf{s}$ is logarithmically and $\mathcal{J}_\mathsf{f}^\mathsf{c}$ is additively bounded, then $\mathtt{irc}_\mathcal{R}(n) \in \mathcal{O}(\log n)$;*

*(ii) if, for all $\mathsf{f} \in \mathcal{F}$, $\mathcal{J}_\mathsf{f}$ is additively bounded, then $\mathtt{irc}_\mathcal{R}(n) \in \mathcal{O}(n)$; and*

*(iii) if, for all defined symbols $\mathsf{f}$ and constructors $\mathsf{c}$, $\mathcal{J}_\mathsf{c}$ is additively and $\mathcal{J}_\mathsf{f}$ is polynomially bounded, then $\mathtt{irc}_\mathcal{R}(n) \in \mathcal{O}\left(n^k\right)$, for some $k \in \mathbb{N}$.*

**Example 4** Let us illustrate this behavior by interpreting functions from Example 1. Interpretation for constructors were given in Example 2.

$$\begin{array}{lcllcl}
\mathcal{J}_{\mathtt{add}} & = & \langle(\boldsymbol{\lambda} xy.y+1), \boldsymbol{\lambda} xy.x+y\rangle & \mathcal{J}_{\mathtt{sum}} & = & \langle(\boldsymbol{\lambda} q.2q_\mathsf{l}+q_\mathsf{l}q_\mathsf{m}), \boldsymbol{\lambda} q.q_\mathsf{l}q_\mathsf{m}\rangle \\
\mathcal{J}_{\mathtt{minus}} & = & \langle(\boldsymbol{\lambda} xy.y+1), \boldsymbol{\lambda} xy.x\rangle & \mathcal{J}_{\mathtt{rev}} & = & \left\langle(\boldsymbol{\lambda} q.q_\mathsf{l}+\frac{q_\mathsf{l}(q_\mathsf{l}+1)}{2}+1), \boldsymbol{\lambda} q.q\right\rangle \\
\mathcal{J}_{\mathtt{quot}} & = & \langle(\boldsymbol{\lambda} xy.x+xy+1), \boldsymbol{\lambda} xy.x\rangle & & & \\
\mathcal{J}_{\mathtt{append}} & = & \langle(\boldsymbol{\lambda} ql.q_\mathsf{l}+1), \boldsymbol{\lambda} ql.\langle q_\mathsf{l}+l_\mathsf{l}, \max(q_\mathsf{m}, q_\mathsf{m})\rangle\rangle & & &
\end{array}$$

Checking compatibility of this interpretation is straightforward. Notice that in each set of rules defining a function $\mathsf{f}$ in Example 1 size components are additively and cost components are polynomially bounded. By case (b) of Lemma 8, we have that $\mathtt{irc}_{\mathcal{R}_{\mathtt{add}}}$, $\mathtt{irc}_{\mathcal{R}_{\mathtt{append}}}$, and $\mathtt{irc}_{\mathcal{R}_{\mathtt{minus}}}$ are linear. Quadratic bounds can be derived to $\mathtt{irc}_{\mathcal{R}_{\mathtt{quot}}}$, $\mathtt{irc}_{\mathcal{R}_{\mathtt{sum}}}$, and $\mathtt{irc}_{\mathcal{R}_{\mathtt{rev}}}$.

Recall the semantic meaning given to size components, see Example 2, one can observe that the cost component of interpretations do not only bound the innermost runtime complexity of $\mathcal{R}_\mathsf{f}$ but also provide additional information on the role each size component plays in the reduction cost. For instance: the cost of adding two numbers depends solely on the size of `add`'s second argument; the cost of summing every element of a list has a linear dependency on its length and non-linear dependency on its length and maximum element. This is particularly useful in program analysis since one can detect a possible costly operation by analyzing the shape of interpretations themselves.

## 5 Automation

In this section, we limn an automation technique implementing a procedure to search for cost–size tuple interpretations.

**A Pseudo-procedure to Search for Cost–Size Tuples.** As it might be expected, tuple interpretations do not provide a complete proof method: there are innermost terminating systems that cannot be oriented. Nevertheless, it has the potential to be very powerful — if we choose the interpretation key $\mathcal{J}_\mathcal{B}$ right. Intuitively, we begin the search by assigning a measure of size to each sort, that is the number $K[\iota]$. In a fully automated setting, where no human input is allowed, all sorts $\iota$ start with $K[\iota] = 1$ and go up to a pre-determined bound $K$.

**Main Procedure**
**Input:** A syntax signature $(\mathcal{B}, \mathcal{F}, \mathtt{ar})$ for a TRS $\mathcal{R}$ together with its set of rules $\ell_i \to r_i$.
**Output:** YES (together with a representation of the cost–size tuple found), if a cost–size tuple could be found; MAYBE, if all steps below where executed and no interpretation could be found[1].

1. Split the signature into two disjoint sets of constructors and defined symbols, i.e., $\mathcal{F} = \mathcal{C} \cup \mathcal{D}$.

[1] Notice that with only this approach we cannot possibly return NO.

2. For each constructor $\mathsf{c} : \iota_1 \Rightarrow \ldots \Rightarrow \iota_m \Rightarrow \kappa$, choose its cost interpretation as the zero-valued cost function $(\boldsymbol{\lambda} x_1 \ldots x_m.0)$; size interpretations are additive.
3. Split $\mathcal{D}$ into sets $\mathcal{D}_1, \ldots, \mathcal{D}_k$ such that for each $\mathsf{f} \in \mathcal{D}_i$, all function symbols occurring in the rules defining $\mathsf{f}$ are either constructors or in $\mathcal{D}_1 \cup \cdots \cup \mathcal{D}_i$.
4. For all $1 \leq i \leq n$, choose an *interpretation shape* for the symbols in $\mathcal{D}_i$ based on the selector strategy $\mathcal{S}$ (to be defined below). Remove the associated choice from $\mathcal{S}$ parametrized by $\mathsf{f}$.
   - If no choice can be made by $\mathcal{S}$ stop and return MAYBE.
5. *Simplify* $[\![\ell]\!] \succ [\![r]\!]$ so that the result is a set of order constraints that does not depend on any interpreted variable (we shall define this simplification step below).
6. *Check* if $C$ holds.
   - If all constraints in $C$ hold, then return YES.
   - Otherwise, increase $K[\iota]$ by one, update the additive size interpretation for the constructors, and return to step 4 choosing another interpretation shape.

**Strategy-based Search for Tuple Interpretations.** Two key aspects of the previous procedure remain to be defined. The strategy $\mathcal{S}$ for selecting interpretation shapes and the *check* command in step 6. Intuitively, a strategy is a procedure that implement choices for interpreting defined symbols in $\mathcal{D}_i$. For instance, we could randomly pick an interpretation shape from a list (the **blind** strategy); we could incrementally select interpretations from a list of possible attempts (the **progressive** strategy); or we could select interpretations based on their syntax patterns (the **pattern** strategy). The *check* procedure depends very much on the type of interpretations and which class of weakly monotonic functions is allowed. We illustrate each strategy and checker in the case where interpretations are polynomials and max-polynomials.

**Definition 13 (Interpretation Shapes)** Let $\sigma = \iota_1 \Rightarrow \ldots \Rightarrow \iota_m \Rightarrow \kappa$, and each $f_{ij}$ appearing in the shapes below is a additively bounded weakly monotonic function over $\mathcal{S}_\sigma$.

- The additive class of interpretations contains additively bounded cost–size functionals of the following form:

$$\boldsymbol{\lambda} x_1 \ldots x_m. \sum_{i=1}^{m} \sum_{j=1}^{K[\iota_i]} x_{ij} + b_0 + b_1 f_1(x_1, \ldots, x_m) + \cdots + b_N f_N(x_1, \ldots, x_m);$$

- The linear class contains cost–size functionals written as:

$$\boldsymbol{\lambda} x_1 \ldots x_m. \sum_{i=1}^{m} \sum_{j=1}^{K[\iota_i]} a_{ij} f_{ij}(x_1, \ldots, x_m) x_{ij};$$

- The simple class contains cost–size functionals written as:

$$\boldsymbol{\lambda} x_1 \ldots x_m. \sum_{i_j \in \{0,1\}} a_{i_1}, \ldots, a_{i_m} f_{i_1}(\vec{x}), \ldots, f_{i_m}(\vec{x}) x_1^{i_1} \cdots x_m^{i_m};$$

- The simple quadratic class contains cost–size functionals written as a sum of a simple functional plus a quadratic component:

$$\boldsymbol{\lambda} x_1 \ldots x_m. \sum_{i_j \in \{0,1\}} a_{i_1} \ldots a_{i_m} f_{i_1}(\vec{x}) \ldots f_{i_m}(\vec{x}) x_1^{i_1} \cdots x_m^{i_m} + \sum_{1 \leq i \leq m} b_i x_i^2;$$

- and finally, the quadratic class contains cost–size functions where we allow general product of variables with degree at maximum 2:

$$\boldsymbol{\lambda} x_1 \dots x_m. \sum_{i_j \in \{0,1,2\}} a_{i_1}, \dots, a_{i_m} f_{i_1}(\vec{x}) \dots f_{i_m}(\vec{x}) x_1^{i_1} \cdots x_m^{i_m}.$$

An interpretation is a max-polynomial if its functional coefficients are composed of compositions of max functions only.

Hence, the blind strategy randomly selects one of the shapes above. The incremental strategy chooses interpretations in order, starting from additive up to quadratic. The pattern strategy is slightly more difficult to realize since we need heuristic analysis on the shape of rules. For instance, every rule of the form $\mathsf{f}\, x_1 \dots x_m \to x_i$ have constant cost functions ($\boldsymbol{\lambda} x_1 \dots x_m.1$) and additive size components. Rules that copy variables, i.e., follow the pattern $C[x] \to D[x,x]$, induce at least quadratic bound on cost. Notice that this is the case for all quadratic complexities in this paper.

In order to simplify constraints $[\![\ell]\!] \succ [\![r]\!]$ we have to simplify inequalities between polynomials (max-polynomials). To simplify polynomial (max-polynomial) shapes, we need to compare polynomials $P^{\mathsf{c}}_{\ell} > R^{\mathsf{c}}_{r}$ and $P^{\mathsf{s}}_{\ell_1} \sqsupseteq P^{\mathsf{s}}_{r_1} \wedge \dots \wedge P^{\mathsf{s}}_{\ell_{K[\tau]}} \sqsupseteq P^{\mathsf{s}}_{r_{K[\kappa]}}$. These conditions are then reduced to formulas in QF_NIA (Quantifier-Free Non-Linear Integer Arithmetic) and sent to an SMT solver, see [9]. Max polynomials are simplified using the rules $\max(x,y)+z \rightsquigarrow \max(x+z,y+z)$ and $\max(x,y)z \rightsquigarrow \max(xz,yz)$. The result has the form $\max_l P_l$ where each $P_l$ is a polynomial without max occurrences [7].

**Work-in-progress Implementation.** Parallel to the theoretical development, we are working on implementing the essential structural codebase to run the **Main Procedure** above. Currently, this is still a prototype, in which we can handle simple interpretation shapes such as additive and linear. See [10] for more details.

## 6 Conclusion

In this paper we showed that cost–size tuple pairs can be adapted to handle innermost rewriting. The type-aware algebraic interpretation style provided the machinery necessary to deal with innermost termination and a mechanism to bound the innermost runtime complexity of compatible TRSs. We presented sufficient conditions for feasible (polynomial) bounds on $\mathtt{irc}_{\mathcal{R}}$ of compatible systems, which are in line with related works on the literature. This line of investigation is far from over. Since searching for interpretations can be cumbersome, our immediate future work is to develop new strategies and interpretation shapes. This has the potential to drastically improve the efficiency of our prototype tool.

**Acknowledgments.** We wish to thank Cynthia Kop — for the valuable discussions and guidance during the production of this paper; we thank Niels van der Weide, Marcos Bueno, and Edna Gomes — for carefully proofread the various manuscript versions of the paper; and we thank the anonymous referees — for the comments that helped improve the paper.

## References

[1] M. Avanzini & G. Moser (2008): *Complexity Analysis by Rewriting*. In: *Proc. FLOPS*, doi:10.1007/978-3-540-78969-7_11.

[2] P. Baillot & U. Dal Lago (2016): *Higher-order interpretations and program complexity*. *IC*, doi:10.1016/j.ic.2015.12.008.

[3] A. Ben Cherifa & P. Lescanne (1987): *Termination of rewriting systems by polynomial interpretations and its implementation*. *Science of Computer Programming* 9(2), pp. 137–159, doi:https://doi.org/10.1016/0167-6423(87)90030-X.

[4] G. Bonfante, A. Cichon, J.-Y. Marion & H. Touzet (2001): *Algorithms with polynomial interpretation termination proof*. *Journal of Functional Programming* 11(1), p. 33–53, doi:10.1017/S0956796800003877.

[5] G. Bonfante, J. Marion & J. Moyen (2001): *On Lexicographic Termination Ordering with Space Bound Certifications*. In: *Proc. PSI*, doi:10.1007/3-540-45575-2_46.

[6] A. Cichon & P. Lescanne (1992): *Polynomial interpretations and the complexity of algorithms*. In: *CADE*, pp. 139–147, doi:10.1007/3-540-55602-8_161.

[7] M. Codish, I. Gonopolskiy, A. M. Ben-Amram, C. Fuhs & J. Giesl (2011): *SAT-based termination analysis using monotonicity constraints over the integers*. *Theory and Practice of Logic Programming* 11(4-5), p. 503–520, doi:https://doi.org10.1017/S1471068411000147.

[8] E. Contejan, C. Marché, A. P. Tomás & X. Urbain (2005): *Mechanically Proving Termination Using Polynomial Interpretations*. *JAR* (34), doi:10.1007/s10817-005-9022-x.

[9] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plucker, P. Schneider-Kamp, T. Stroder, S. Swiderski & R. Thiemann (2017): *Analyzing Program Termination and Complexity Automatically with AProVE*. *JAR* (58), pp. 3–31, doi:10.1007/s10817-016-9388-y.

[10] L. Guo & D. Vale (2022): *Hermes: Innermost Runtime Complexity Analysis Tool*. Software Repository. Available at `https://github.com/deividrvale/hermes`.

[11] N. Hirokawa & G. Moser (2008): *Automated Complexity Analysis Based on the Dependency Pair Method*. In: *Proc. IJCAR*, doi:10.1007/978-3-540-71070-7_32.

[12] D. Hofbauer (1992): *Termination proofs by multiset path orderings imply primitive recursive derivation lengths*. *Proc. TCS*, doi:10.1007/3-540-53162-9_50.

[13] D. Hofbauer (2001): *Termination Proofs by Context-Dependent Interpretations*. In: *Proc. RTA*, doi:10.1007/3-540-45127-7_10.

[14] D. Hofbauer & C. Lautemann (1989): *Termination proofs and the length of derivations*. In: *Proc. RTA*, doi:10.1007/3-540-51081-8_107.

[15] G. Huet & D.C Oppen (1980): *Equations and rewrite rules: a survey*. Formal Language Theory: Perspectives and Open Problems. Available at `http://rewriting.loria.fr/documents/CS-TR-80-785.pdf`.

[16] C. Kop & D. Vale (2021): *Tuple Interpretations for Higher-Order Complexity*. In: *FSCD*, pp. 31:1–31:22, doi:10.4230/LIPIcs.FSCD.2021.31.

[17] C. Lautemann (1988): *A note on polynomial interpretation*. *Bulletin EATCS* volume 4, pp. 129–131.

[18] F. Mitterwallner & A. Middeldorp (2022): *Polynomial Termination Over $\mathbb{N}$ Is Undecidable*. In: *Proc. FSCD*, pp. 27:1–27:17, doi:https://doi.org/10.4230/LIPIcs.FSCD.2022.27.

[19] G. Moser (2017): *Uniform Resource Analysis by Rewriting: Strengths and Weaknesses (Invited Talk)*. In: *Proc. FSCD*, pp. 2:1–2:10, doi:10.4230/LIPIcs.FSCD.2017.2.

[20] G. Moser, A. Schnabl & J. Waldmann (2008): *Complexity Analysis of Term Rewriting Based on Matrix and Context Dependent Interpretations*. In: *Proc. IARCS*, pp. 304–315, doi:10.4230/LIPIcs.FSTTCS.2008.1762.

[21] L. Noschinski, F. Emmes & J. Giesl (2011): *A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems*. In: *CADE-23*, pp. 422–438, doi:10.1007/978-3-642-22438-6_32.

[22] Yoshihito T. (1987): *Counterexamples to termination for the direct sum of term rewriting systems*. *Information Processing Letters* 25(3), pp. 141–143, doi:https://doi.org/10.1016/0020-0190(87)90122-0.

[23] A. Weiermann (1995): *Termination proofs for term rewriting systems by lexicographic path orderings imply multiply recursive derivation lengths*. *TCS*, doi:10.1016/0304-3975(94)00135-6.

[24] A. Yamada (2022): *Tuple Interpretations for Termination of Term Rewriting*. J Autom Reasoning, doi:https://doi.org/10.1007s10817-022-09640-4.

# Equational Theorem Proving for Clauses over Strings

Dohan Kim
A. I. Research Lab, Kyungwon Plaza 201, Sujeong-gu, Seongnam-si, Gyeonggi-do, South Korea
dkim@airesearch.kr

Although reasoning about equations over strings has been extensively studied for several decades, little research has been done for equational reasoning on general clauses over strings. This paper introduces a new superposition calculus with strings and present an equational theorem proving framework for clauses over strings. It provides a saturation procedure for clauses over strings and show that the proposed superposition calculus with contraction rules is refutationally complete. This paper also presents a new decision procedure for word problems over strings w.r.t. a set of conditional equations $R$ over strings if $R$ can be finitely saturated under the proposed inference system.

## 1 Introduction

Strings are fundamental objects in mathematics and many fields of science including computer science and biology. Reasoning about equations over strings has been widely studied in the context of string rewriting systems, formal language theory, word problems in semigroups, monoids and groups [8, 14], etc. Roughly speaking, reasoning about equations over strings replaces equals by equals w.r.t. a given reduction ordering $\succ$. For example, if we have two equations over strings $u_1u_2u_3 \approx s$ and $u_2 \approx t$ with $u_1u_2u_3 \succ s$ and $u_2 \succ t$, where $u_2$ is not the empty string, then we may infer the equation $u_1tu_3 \approx s$ by replacing $u_2$ in $u_1u_2u_3 \approx s$ with $t$. Meanwhile, if we have two equations over strings $u_1u_2 \approx s$ and $u_2u_3 \approx t$ with $u_1u_2 \succ s$ and $u_2u_3 \succ t$, where $u_2$ is not the empty string, then we should also be able to infer the equation $u_1t \approx su_3$. This can be done by concatenating $u_3$ to both sides of $u_1u_2 \approx s$ (i.e., $u_1u_2u_3 \approx su_3$) and then replacing $u_2u_3$ in $u_1u_2u_3 \approx su_3$ with $t$. Here, the *monotonicity property* of equations over strings is assumed, i.e., $s \approx t$ implies $usv \approx utv$ for strings $s$, $t$, $u$, and $v$.[1]

This reasoning about equations over strings is the basic ingredient for *completion* [8, 16] of string rewriting systems. A completion procedure [8, 16] attempts to construct a finite convergent string rewriting system, where a finite convergent string rewriting system provides a decision procedure for its corresponding equational theory.

Unlike reasoning about equations over strings, equational reasoning on general clauses over strings has not been well studied, where clauses are often the essential building blocks for logical statements.

This paper proposes a superposition calculus and an equational theorem proving procedure with clauses over strings. The results presented here generalize the results about completion of equations over strings [8, 16]. Throughout this paper, the monotonicity property of equations over strings is assumed and considered in the proposed inference rules. This assumption is natural and common to equations over strings occurring in algebraic structures (e.g., semigroups and monoids), formal language theory, etc. The *cancellation property* of equations over strings is not assumed, i.e., $su \approx tu$ implies $s \approx t$ for strings $s$, $t$, and a nonempty string $u$ (cf. *non-cancellative* [8] algebraic structures).

Now, the proposed superposition inference rule is given roughly as follows:

[1] Note that it suffices to assume the right monotonicity property of equations over strings, i.e., $s \approx t$ implies $su \approx tu$ for strings $s$, $t$, and $u$, when finding overlaps between equations over strings under the monotonicity assumption.

**Superposition:** $\dfrac{C \vee u_1u_2 \approx s \qquad D \vee u_2u_3 \approx t}{C \vee D \vee u_1t \approx su_3}$

if $u_2$ is not the empty string, and $u_1u_2 \succ s$ and $u_2u_3 \succ t$.

Intuitively speaking, using the monotonicity property, $C \vee u_1u_2u_3 \approx su_3$ can be obtained from the left premise $C \vee u_1u_2 \approx s$. Then the above inference by Superposition can be viewed as an application of a conditional rewrite rule $D \vee u_2u_3 \approx t$ to $C \vee u_1u_2u_3 \approx su_3$, where $u_2u_3$ in $C \vee u_1u_2u_3 \approx su_3$ is now replaced by $t$, and $D$ is appended to the conclusion. (Here, $D$ can be viewed as consisting of the positive and negative conditions.) Note that both $u_1$ and $u_3$ can be the empty string in the Superposition inference rule. These steps are combined into a single Superposition inference step. For example, suppose that we have three clauses 1: $ab \approx d$, 2: $bc \approx e$, and 3: $ae \not\approx dc$. We use the Superposition inference rule with 1 and 2, and obtain 4: $ae \approx dc$ from which we derive a contradiction with 3. The details of the inference rules in the proposed inference system are discussed in Section 3.

The proposed superposition calculus is based on the simple string matching methods and the efficient length-lexicographic ordering instead of using equational unification and the more complex orderings, such as the lexicographic path ordering (LPO) [13] and Knuth-Bendix ordering (KBO) [2].

This paper shows that a clause over strings can be translated into a clause over first-order terms, which allows one to use the existing notion of redundancy in the literature [3, 22] for clauses over strings. Based on the notion of redundancy, one may delete redundant clauses using the contraction rules (i.e., Simplification, Subsumption, and Tautology) during an equational theorem proving derivation in order to reduce the search space for a refutation.

The *model construction techniques* [3, 22] is adapted for the refutational completeness of the proposed superposition calculus. This paper also uses a Herbrand interpretation by translating clauses over strings into clauses over first-order terms, where each nonground first-order clause represents all its ground instances. Note that this translation is not needed for the proposed inference system itself.

Finally, the proposed equational theorem proving framework with clauses over strings allows one to provide a new decision procedure for word problems over strings w.r.t. a conditional equational theory $R$ if $R$ can be finitely saturated under the proposed inference system.

## 2 Preliminaries

It is assumed that the reader has some familiarity with equational theorem proving [3, 22] and string rewriting systems [8, 16, 17]. The notion of conditional equations and Horn clauses are discussed in [12].

An *alphabet* $\Sigma$ is a finite set of symbols (or letters). The set of all strings of symbols over $\Sigma$ is denoted $\Sigma^*$ with the empty string $\lambda$.

If $s \in \Sigma^*$, then the *length* of $s$, denoted $|s|$, is defined as follows: $|\lambda| := 0$, $|a| := 1$ for each $a \in \Sigma$, and $|sa| := |s| + 1$ for $s \in \Sigma^*$ and $a \in \Sigma$.

A *multiset* is an unordered collection with possible duplicate elements. We denote by $M(x)$ the number of occurrences of an object $x$ in a multiset $M$.

An *equation* is an expression $s \approx t$, where $s$ and $t$ are strings, i.e., $s,t \in \Sigma^*$. A *literal* is either a positive equation $L$, called a *positive literal*, or a negative equation $\neg L$, called a *negative literal*. We also write a negative literal $\neg(s \approx t)$ as $s \not\approx t$. We identify a positive literal $s \approx t$ with the multiset $\{\{s\},\{t\}\}$ and a negative literal $s \not\approx t$ with the multiset $\{\{s,t\}\}$. A *clause* (over $\Sigma^*$) is a finite multiset of literals, written as a disjunction of literals $\neg A_1 \vee \cdots \vee \neg A_m \vee B_1 \vee \cdots \vee B_n$ or as an implication $\Gamma \rightarrow \Delta$, where

$\Gamma = A_1 \wedge \cdots \wedge A_m$ and $\Delta = B_1 \vee \cdots \vee B_n$. We say that $\Gamma$ is the *antecedent* and $\Delta$ is the *succedent* of clause $\Gamma \to \Delta$. A *Horn clause* is a clause with at most one positive literal. The *empty clause*, denoted $\Box$, is the clause containing no literals.

A *conditional equation* is a clause of the form $(s_1 \approx t_1 \wedge \cdots \wedge s_n \approx t_n) \to l \approx r$. If $n = 0$, a conditional equation is simply an equation. A conditional equation is naturally represented by a Horn clause. A *conditional equational theory* is a set of conditional equations.

Any ordering $\succ_S$ on a set $S$ can be extended to an ordering $\succ_S^{mul}$ on finite multisets over $S$ as follows: $M \succ_S^{mul} N$ if (i) $M \neq N$ and (ii) whenever $N(x) > M(x)$ then $M(y) > N(y)$, for some $y$ such that $y \succ_S x$.

Given a multiset $M$ and an ordering $\succ$ on $M$, we say that $x$ is *maximal* (resp. *strictly maximal*) in $M$ if there is no $y \in M$ (resp. $y \in M \setminus \{x\}$) with $y \succ x$ (resp. $y \succ x$ or $x = y$).

An ordering $>$ on $\Sigma^*$ is *terminating* if there is no infinite chain of strings $s > s_1 > s_2 > \cdots$ for any $s \in \Sigma^*$. An ordering $>$ on $\Sigma^*$ is *admissible* if $u > v$ implies $xuy > xvy$ for all $u, v, x, y \in \Sigma^*$. An ordering $>$ on $\Sigma^*$ is a *reduction ordering* if it is terminating and admissible.

The *lexicographic ordering* $\succ_{lex}$ induced by a total precedence ordering $\succ_{prec}$ on $\Sigma$ ranks strings of the same length in $\Sigma^*$ by comparing the letters in the first index position where two strings differ using $\succ_{prec}$. For example, if $a = a_1 a_2 \cdots a_k$ and $b = b_1 b_2 \cdots b_k$, and the first index position where $a$ and $b$ are differ is $i$, then $a \succ_{lex} b$ if and only if $a_i \succ_{prec} b_i$.

The *length-lexicographic ordering* $\succ$ on $\Sigma^*$ is defined as follows: $s \succ t$ if and only if $|s| > |t|$, or they have the same length and $s \succ_{lex} t$ for $s, t \in \Sigma^*$. If $\Sigma$ and $\succ_{prec}$ are fixed, then it is easy to see that we can determine whether $s \succ t$ for two (finite) input strings $s \in \Sigma^*$ and $t \in \Sigma^*$ in $O(n)$ time, where $n = |s| + |t|$. The length-lexicographic ordering $\succ$ on $\Sigma^*$ is a reduction ordering. We also write $\succ$ for a multiset extension of $\succ$ if it is clear from context.

We say that $\approx$ has the *monotonicity property* over $\Sigma^*$ if $s \approx t$ implies $usv \approx utv$ for all $s, t, u, v \in \Sigma^*$. Throughout this paper, it is assumed that $\approx$ has the monotonicity property over $\Sigma^*$.

# 3 Superposition with Strings

## 3.1 Inference Rules

The following inference rules for clauses over strings are parameterized by a selection function $\mathscr{S}$ and the length-lexicographic ordering $\succ$, where $\mathscr{S}$ arbitrarily selects exactly one negative literal for each clause containing at least one negative literal (see Section 3.6 in [22] or Section 6 in [5]). In this strategy, an inference involving a clause with a selected literal is performed before an inference from clauses without a selected literal for a theorem proving process. The intuition behind the (eager) selection of negative literals is that, roughly speaking, one may first prove the whole antecedent of a clause from other clauses. Then clauses with no selected literals are involved in the main deduction process. This strategy is particularly useful when we consider Horn completion in Section 6 and a decision procedure for the word problems associated with it. In the following, the symbol $\bowtie$ is used to denote either $\approx$ or $\not\approx$.

**Superposition:** $$\frac{C \vee u_1 u_2 \approx s \qquad D \vee u_2 u_3 \approx t}{C \vee D \vee u_1 t \approx s u_3}$$

if (i) $u_2$ is not $\lambda$, (ii) $C$ contains no selected literal, (iii) $D$ contains no selected literal, (iv) $u_1 u_2 \succ s$, and (v) $u_2 u_3 \succ t$.[2]

[2]We do not require that $u_1 u_2 \approx s$ (resp. $u_2 u_3 \approx t$) is strictly maximal in the left premise (resp. the right premise) because of the assumption on the monotonicity property of equations over strings (see also Lemma 1 in Section 3.2).

**Rewrite:** $\dfrac{C \vee u_1u_2u_3 \bowtie s \qquad D \vee u_2 \approx t}{C \vee D \vee u_1tu_3 \bowtie s}$

if (i) $u_1u_2u_3 \bowtie s$ is selected for the left premise whenever $\bowtie$ is $\not\approx$, (ii) $C$ contains no selected literal whenever $\bowtie$ is $\approx$, (iii) $D$ contains no selected literal, and (iv) $u_2 \succ t$.[3]

**Equality Resolution:** $\dfrac{C \vee s \not\approx s}{C}$

if $s \not\approx s$ is selected for the premise.

The following Paramodulation and Factoring inference rules are used for non-Horn clauses containing positive literals only (cf. *Equality Factoring* [3, 22] and *Merging Paramodulation* rule [3]).

**Paramodulation:** $\dfrac{C \vee s \approx u_1u_2 \qquad D \vee u_2u_3 \approx t}{C \vee D \vee su_3 \approx u_1t}$

if (i) $u_2$ is not $\lambda$, (ii) $C$ contains no selected literal, (iii) $C$ contains a positive literal, (iv) $D$ contains no selected literal, (v) $s \succ u_1u_2$, and (vi) $u_2u_3 \succ t$.

**Factoring:** $\dfrac{C \vee s \approx t \vee su \approx tu}{C \vee su \approx tu}$

if $C$ contains no selected literal.

In the proposed inference system, finding whether a string $s$ occurs within a string $t$ can be done in linear time in the size of $s$ and $t$ by using the existing string matching algorithms such as the Knuth-Morris-Pratt (KMP) algorithm [9]. For example, the KMP algorithm can be used for finding $u_2$ in $u_1u_2u_3$ in the Rewrite rule and finding $u_2$ in $u_1u_2$ in the Superposition and Paramodulation rule.

In the remainder of this paper, we denote by $\mathfrak{S}$ the inference system consisting of the Superposition, Rewrite, Equality Resolution, Paramodulation and the Factoring rule, and denote by $S$ a set of clauses over strings. Also, by the *contraction rules* we mean the following inference rules–Simplification, Subsumption and Tautology.

**Simplification:** $\dfrac{S \cup \{C \vee l_1ll_2 \bowtie v,\ l \approx r\}}{S \cup \{C \vee l_1rl_2 \bowtie v,\ l \approx r\}}$

if (i) $l_1ll_2 \bowtie v$ is selected for $C \vee l_1ll_2 \bowtie v$ whenever $\bowtie$ is $\not\approx$, (ii) $l_1$ is not $\lambda$, and (iii) $l \succ r$.

In the following inference rule, we say that a clause $C$ *subsumes* a clause $C'$ if $C$ is contained in $C'$, where $C$ and $C'$ are viewed as the finite multisets.

**Subsumption:** $\dfrac{S \cup \{C, C'\}}{S \cup \{C\}}$

if $C \subseteq C'$.

**Tautology:** $\dfrac{S \cup \{C \vee s \approx s\}}{S}$

**Example 1.** Let $a \succ b \succ c \succ d \succ e$ and consider the following inconsistent set of clauses 1: $ad \approx b \vee ad \approx c$, 2: $b \approx c$, 3: $ad \approx e$, and 4: $c \not\approx e$. Now, we show how the empty clause is derived:

[3]Note that $u_2 \succ t$ implies that $u_2$ cannot be the empty string $\lambda$.

5: $ad \approx c \vee ad \approx c$ (Paramodulation of 1 with 2)
6: $ad \approx c$ (Factoring of 5)
7: $c \approx e$ (Rewrite of 6 with 3)
8: $e \not\approx e$ ($c \not\approx e$ is selected for 4. Rewrite of 4 with 7)
9: $\Box$ ($e \not\approx e$ is selected for 8. Equality Resolution on 8)

Note that there is no inference with the selected literal in 4 from the initial set of clauses 1, 2, 3, and 4. We produced clauses 5, 6, and 7 without using a selected literal. Once we have clause 7, there is an inference with the selected literal in 4.

**Example 2.** Let $a \succ b \succ c \succ d$ and consider the following inconsistent set of clauses 1: $aa \approx a \vee bd \not\approx a$, 2: $cd \approx b$, 3: $ad \approx c$, 4: $bd \approx a$, and 5: $dab \not\approx db$. Now, we show how the empty clause is derived:
6: $aa \approx a \vee a \not\approx a$ ($bd \not\approx a$ is selected for 1. Rewrite of 1 with 4)
7: $aa \approx a$ ($a \not\approx a$ is selected for 6. Equality resolution on 6)
8: $ac \approx ad$ (Superposition of 7 with 3)
9: $add \approx ab$ (Superposition of 8 with 2)
10: $ab \approx cd$ (Rewrite of 9 with 3)
11: $dcd \not\approx db$ ($dab \not\approx db$ is selected for 5. Rewrite of 5 with 10)
12: $db \not\approx db$ ($dcd \not\approx db$ is selected for 11. Rewrite of 11 with 2)
13: $\Box$ ($db \not\approx db$ is selected for 12. Equality Resolution on 12)

### 3.2 Lifting Properties

Recall that $\Sigma^*$ is the set of all strings over $\Sigma$ with the empty string $\lambda$. We let $T(\Sigma \cup \{\bot\})$ be the set of all first-order ground terms over $\Sigma \cup \{\bot\}$, where each letter from $\Sigma$ is interpreted as a unary function symbol and $\bot$ is the only constant symbol. (The constant symbol $\bot$ does not have a special meaning (e.g., "false") in this paper.) We remove parentheses for notational convenience for each term in $T(\Sigma \cup \{\bot\})$. Since $\bot$ is the only constant symbol, we see that $\bot$ occurs only once at the end of each term in $T(\Sigma \cup \{\bot\})$. We may view each term in $T(\Sigma \cup \{\bot\})$ as a string ending with $\bot$. Now, the definitions used in Section 2 can be carried over to the case when $\Sigma^*$ is replaced by $T(\Sigma \cup \{\bot\})$. In the remainder of this paper, we use the string notation for terms in $T(\Sigma \cup \{\bot\})$ unless otherwise stated.

Let $s \approx t$ be an equation over $\Sigma^*$. Then we can associate $s \approx t$ with the equation $s(x) \approx t(x)$, where $s(x) \approx t(x)$ represents the set of all its ground instances over $T(\Sigma \cup \{\bot\})$. (Here, $\lambda(x)$ and $\lambda\bot$ correspond to $x$ and $\bot$, respectively.) First, $s \approx t$ over $\Sigma^*$ corresponds to $s\bot \approx t\bot$ over $T(\Sigma \cup \{\bot\})$. Now, using the monotonicity property, if we concatenate string $u$ to both sides of $s \approx t$ over $\Sigma^*$, then we have $su \approx tu$, which corresponds to $su\bot \approx tu\bot$.

There is a similar approach in string rewriting systems. If $S$ is a string rewriting system over $\Sigma^*$, then it is known that we can associate term rewriting system $R_S$ with $S$ in such a way that $R_S := \{l(x) \to r(x) \mid l \to r \in S\}$ [8], where $x$ is a variable and each letter from $\Sigma$ is interpreted as a unary function symbol. We may rename variables (by standardizing variables apart) whenever necessary. This approach is particularly useful when we consider critical pairs between the rules in a string rewriting system. For example, if there are two rules $aa \to c$ and $ab \to d$ in $S$, then we have $cb \leftarrow aab \to ad$, where $<cb, ad>$ (or $<ad, cb>$) is a *critical pair* formed from these two rules. This critical pair can also be found if we associate $aa \to c \in S$ with $a(a(x)) \to c(x) \in R_S$ and $ab \to d \in S$ with $a(b(x)) \to d(x) \in R_S$. First, we rename the rule $a(b(x)) \to d(x) \in R_S$ into $a(b(y)) \to d(y)$. Then by mapping $x$ to $b(z)$ and $y$ to $z$, we have $c(b(z)) \leftarrow a(a(b(z))) \to a(d(z))$, where $<c(b(z)), a(d(z))>$ is a critical pair formed from these two rules. This critical pair can be associated with the critical pair $<cb, ad>$ formed from $aa \to c$ in $S$ and

$ab \to d$ in $S$.

However, if $s \not\approx t$ is a negative literal over strings, then we cannot simply associate $s \not\approx t$ with the negative literal $s(x) \not\approx t(x)$ over first-order terms. Suppose to the contrary that we associate $s \not\approx t$ with $s(x) \not\approx t(x)$. Then $s \not\approx t$ implies $su \not\approx tu$ for a nonempty string $u$ because we can substitute $u(y)$ for $x$ in $s(x) \not\approx t(x)$, and $su \not\approx tu$ can also be associated with $s(u(y)) \not\approx t(u(y))$. Using the contrapositive argument, this means that $su \approx tu$ implies $s \approx t$ for the nonempty string $u$. Recall that we do not assume the cancellation property of equations over strings in this paper.[4] Instead, we simply associate $s \not\approx t$ with $s\bot \not\approx t\bot$. The following lemma is based on the above observations. We denote by $T(\Sigma \cup \{\bot\}, X)$ the set of first-order terms built on $\Sigma \cup \{\bot\}$ and a denumerable set of variables $X$, where each symbol from $\Sigma$ is interpreted as a unary function symbol and $\bot$ is the only constant symbol.

**Lemma 1.** *Let $C := s_1 \approx t_1 \vee \cdots \vee s_m \approx t_m \vee u_1 \not\approx v_1 \vee \cdots \vee u_n \not\approx v_n$ be a clause over $\Sigma^*$ and $P$ be the set of all clauses that follow from $C$ using the monotonicity property. Let $Q$ be the set of all ground instances of the clause $s_1(x_1) \approx t_1(x_1) \vee \cdots \vee s_m(x_m) \approx t_m(x_m) \vee u_1\bot \not\approx v_1\bot \vee \cdots \vee u_n\bot \not\approx v_n\bot$ over $T(\Sigma \cup \{\bot\}, X)$, where $x_1, \ldots, x_m$ are distinct variables in $X$ and each letter from $\Sigma$ is interpreted as a unary function symbol. Then there is a one-to-one correspondence between $P$ and $Q$.*

*Proof.* For each element $D$ of $P$, $D$ has the form $D := s_1w_1 \approx t_1w_1 \vee \cdots \vee s_mw_m \approx t_mw_m \vee u_1 \not\approx v_1 \vee \cdots \vee u_n \not\approx v_n$ for some $w_1, \ldots, w_m \in \Sigma^*$. (If $w_i = \lambda$ for all $1 \leq i \leq m$, then $D$ is simply $C$.) Now, we map each element $D$ of $P$ to $D'$ in $Q$, where $D' := s_1w_1\bot \approx t_1w_1\bot \vee \cdots \vee s_mw_m\bot \approx t_mw_m\bot \vee u_1\bot \not\approx v_1\bot \vee \cdots \vee u_n\bot \not\approx v_n\bot$. Since $\bot$ is the only constant symbol in $\Sigma \cup \{\bot\}$, it is easy to see that this mapping is well-defined and bijective. □

**Definition 2.** (i) We say that every term in $T(\Sigma \cup \{\bot\})$ is a *g-term*. (Recall that we remove parentheses for notational convenience.)
(ii) Let $s \approx t$ (resp. $s \to t$) be an equation (resp. a rule) over $\Sigma^*$. We say that $su\bot \approx tu\bot$ (resp. $su\bot \to tu\bot$) for some string $u$ is a *g-equation* (resp. a *g-rule*) of $s \approx t$ (resp. $s \to t$).
(iii) Let $s \not\approx t$ be a negative literal over $\Sigma^*$. We say that $s\bot \not\approx t\bot$ is a (negative) *g-literal* of $s \not\approx t$.
(iv) Let $C := s_1 \approx t_1 \vee \cdots \vee s_m \approx t_m \vee u_1 \not\approx v_1 \vee \cdots \vee u_n \not\approx v_n$ be a clause over $\Sigma^*$. We say that $s_1w_1\bot \approx t_1w_1\bot \vee \cdots \vee s_mw_m\bot \approx t_mw_m\bot \vee u_1\bot \not\approx v_1\bot \vee \cdots \vee u_n\bot \not\approx v_n\bot$ for some strings $w_1, \ldots, w_m$ is a *g-clause* of clause $C$. Here, each $w_k\bot \in T(\Sigma \cup \{\bot\})$ for nonempty string $w_k$ in the $g$-clause is said to be a *substitution part* of $C$.
(v) Let $\pi$ be an inference (w.r.t. $\mathfrak{S}$) with premises $C_1, \ldots, C_k$ and conclusion $D$. Then a *g-instance* of $\pi$ is an inference (w.r.t. $\mathfrak{S}$) with premises $C'_1, \ldots, C'_k$ and conclusion $D'$, where $C'_1, \ldots, C'_k$ and $D'$ are $g$-clauses of $C_1, \ldots, C_k$ and $D$, respectively.

Since each term in $T(\Sigma \cup \{\bot\})$ is viewed as a string, we may consider inferences between $g$-clauses using $\mathfrak{S}$. Note that concatenating a (nonempty) string at the end of a $g$-term is not allowed for any $g$-term over $T(\Sigma \cup \{\bot\})$. For example, $abc\bot d$ is not a $g$-term, and $a\bot \not\approx b\bot \vee abc\bot d \approx def\bot d$ is not a $g$-clause. We emphasize that we are only concerned with inferences between (legitimate) $g$-clauses here.

We may also use the length-lexicographic ordering $\succ_g$ on $g$-terms. Given a total precedence ordering on $\Sigma \cup \{\bot\}$ for which $\bot$ is minimal, it can be easily verified that $\succ_g$ is a total reduction ordering on $T(\Sigma \cup \{\bot\})$. We simply denote the multiset extension $\succ_g^{mul}$ of $\succ_g$ as $\succ_g$ for notational convenience. Similarly, we denote ambiguously all orderings on $g$-terms, $g$-equations, and $g$-clauses over $T(\Sigma \cup \{\bot\})$ by $\succ_g$. Now, we consider the lifting of inferences of $\mathfrak{S}$ between $g$-clauses over $T(\Sigma \cup \{\bot\})$ to inferences of $\mathfrak{S}$ between clauses over $\Sigma^*$. Let $C_1, \ldots, C_n$ be clauses over $\Sigma^*$ and let

[4] One may assume the cancellation property and associate $s \not\approx t$ over strings with $s(x) \not\approx t(x)$ over first-order terms, which is beyond the scope of this paper.

$$\frac{C'_1 \dots C'_n}{C'}$$

be an inference between their $g$-clauses, where $C'_i$ is a $g$-clause of $C_i$ for all $1 \leq i \leq n$. We say that this inference between $g$-clauses can be *lifted* if there is an inference

$$\frac{C_1 \dots C_n}{C}$$

such that $C'$ is a $g$-clause of $C$. In what follows, we assume that a $g$-literal $L'_i$ in $C'_i$ is selected in the same way as $L_i$ in $C_i$, where $L_i$ is a negative literal in $C_i$ and $L'_i$ is a $g$-literal of $L_i$.

Lifting of an inference between $g$-clauses is possible if it does not correspond to a $g$-instance of an inference (w.r.t. $\mathfrak{S}$) into a substitution part of a clause, which is not necessary (see [4, 22]). Suppose that there is an inference between $g$-clauses $C'_1 \dots C'_n$ with conclusion $C'$ and there is also an inference between clauses $C_1 \dots C_n$ over $\Sigma^*$ with conclusion $C$, where $C'_i$ is a $g$-clause of $C_i$ for all $1 \leq i \leq n$. Then, the inference between $g$-clauses $C'_1 \dots C'_n$ over $T(\Sigma \cup \{\bot\})$ can be lifted to the inference between clauses $C_1 \dots C_n$ over $\Sigma^*$ in such a way that $C'$ is a $g$-clause of $C$. This can be easily verified for each inference rule in $\mathfrak{S}$.

**Example 3.** Consider the following Superposition inference with $g$-clauses:

$$\frac{ad\bot \approx cd\bot \vee aabb\bot \approx cbb\bot \qquad abb\bot \approx db\bot}{ad\bot \approx cd\bot \vee adb\bot \approx cbb\bot}$$

where $ad\bot \approx cd\bot \vee aabb\bot \approx cbb\bot$ (resp. $abb\bot \approx db\bot$) is a $g$-clause of $a \approx c \vee aa \approx c$ (resp. $ab \approx d$) and $aabb\bot \succ_g cbb\bot$ (resp. $abb\bot \succ_g db\bot$). This Superposition inference between $g$-clauses can be lifted to the following Superposition inference between clauses over $\Sigma^*$:

$$\frac{a \approx c \vee aa \approx c \qquad ab \approx d}{a \approx c \vee ad \approx cb}$$

where $aa \succ c$ and $ab \succ d$. We see that conclusion $ad\bot \approx cd\bot \vee adb\bot \approx cbb\bot$ of the Superposition inference between the above $g$-clauses is a $g$-clause of conclusion $a \approx c \vee ad \approx cb$ of this inference.

**Example 4.** Consider the following Rewrite inference with $g$-clauses:

$$\frac{a\bot \not\approx d\bot \vee aabb\bot \not\approx cd\bot \qquad abb\bot \approx cb\bot}{a\bot \not\approx d\bot \vee acb\bot \not\approx cd\bot}$$

where $aabb\bot \not\approx cd\bot$ is selected and $a\bot \not\approx d\bot \vee aabb\bot \not\approx cd\bot$ (resp. $abb\bot \approx cb\bot$) is a $g$-clause of $a \not\approx d \vee aabb \not\approx cd$ (resp. $ab \approx c$) with $abb\bot \succ_g cb\bot$. This Rewrite inference between $g$-clauses can be lifted to the following Rewrite inference between clauses over $\Sigma^*$:

$$\frac{a \not\approx d \vee aabb \not\approx cd \qquad ab \approx c}{a \not\approx d \vee acb \not\approx cd}$$

where $aabb \not\approx cd$ is selected and $ab \succ c$. We see that conclusion $a\bot \not\approx d\bot \vee acb\bot \not\approx cd\bot$ of the Rewrite inference between the above $g$-clauses is a $g$-clause of conclusion $a \not\approx d \vee acb \not\approx cd$ of this inference.

## 4 Redundancy and Contraction Techniques

By Lemma 1 and Definition 2, we may translate a clause $C := s_1 \approx t_1 \vee \dots \vee s_m \approx t_m \vee u_1 \not\approx v_1 \vee \dots \vee u_n \not\approx v_n$ over $\Sigma^*$ with all its implied clauses using the monotonicity property into the clause $s_1(x_1) \approx t_1(x_1) \vee \dots \vee s_m(x_m) \approx t_m(x_m) \vee u_1\bot \not\approx v_1\bot \vee \dots \vee u_n\bot \not\approx v_n\bot$ over $T(\Sigma \cup \{\bot\}, X)$ with all its ground instances, where $x_1, \dots, x_m$ are distinct variables in $X$, each symbol from $\Sigma$ is interpreted as a unary function symbol, and $\bot$ is the only constant symbol. This allows us to adapt the existing notion of redundancy found in the literature [3, 22].

**Definition 3.** (i) Let $R$ be a set of $g$-equations or $g$-rules. Then the congruence $\leftrightarrow_R^*$ defines an equality *Herbrand Interpretation* $I$, where the domain of $I$ is $T(\Sigma \cup \{\bot\})$. Each unary function symbol $s \in \Sigma$ is interpreted as the unary function $s_I$, where $s_I(u\bot)$ is the $g$-term $su\bot$. (The constant symbol $\bot$ is simply interpreted as the constant $\bot$.) The only predicate $\approx$ is interpreted by $s\bot \approx t\bot$ if $s\bot \leftrightarrow_R^* t\bot$. We denote by $R^*$ the interpretation $I$ defined by $R$ in this way. $I$ *satisfies* (is a *model* of) a $g$-clause $\Gamma \to \Delta$, denoted by $I \models \Gamma \to \Delta$, if $I \not\supseteq \Gamma$ or $I \cap \Delta \neq \emptyset$. In this case, we say that $\Gamma \to \Delta$ is *true* in $I$. We say that $I$ *satisfies* a clause $C$ over $\Sigma^*$ if $I$ satisfies all $g$-clauses of $C$. We say that $I$ *satisfies* a set of clauses $S$ over $\Sigma^*$, denoted by $I \models S$, if $I$ satisfies every clause in $S$.
(ii) A $g$-clause $C$ *follows* from a set of $g$-clauses $\{C_1, \ldots, C_n\}$, denoted by $\{C_1, \ldots, C_n\} \models C$, if $C$ is true in every model of $\{C_1, \ldots, C_k\}$.

**Definition 4.** Let $S$ be a set of clauses over $\Sigma^*$.
(i) A $g$-clause $C$ is *redundant* w.r.t. $S$ if there exist $g$-clauses $C'_1, \ldots, C'_k$ of clauses $C_1, \ldots, C_k$ in $S$, such that $\{C'_1, \ldots, C'_k\} \models C$ and $C \succ_g C'_i$ for all $1 \leq i \leq k$. A clause in $S$ is *redundant* w.r.t. $S$ if all its $g$-clauses are redundant w.r.t. $S$.
(ii) An inference $\pi$ with conclusion $D$ is *redundant* w.r.t. $S$ if for every $g$-instance of $\pi$ with maximal premise $C'$ (w.r.t. $\succ_g$) and conclusion $D'$, there exist $g$-clauses $C'_1, \ldots, C'_k$ of clauses $C_1, \ldots, C_k$ in $S$ such that $\{C'_1, \ldots, C'_k\} \models D'$ and $C' \succ_g C'_i$ for all $1 \leq i \leq k$, where $D'$ is a $g$-clause of $D$.

**Lemma 5.** *If an equation $l \approx r$ simplifies a clause $C \vee l_1 l l_2 \bowtie v$ into $C \vee l_1 r l_2 \bowtie v$ using the Simplification rule, then $C \vee l_1 l l_2 \bowtie v$ is redundant w.r.t. $\{C \vee l_1 r l_2 \bowtie v, l \approx r\}$.*

*Proof.* Suppose that $l \approx r$ simplifies $D := C \vee l_1 l l_2 \not\approx v$ into $C \vee l_1 r l_2 \not\approx v$, where $l_1 l l_2 \not\approx v$ is selected for $D$. Then, every $g$-clause $D'$ of $D$ has the form $D' := C' \vee l_1 l l_2 \bot \not\approx v\bot$, where $C'$ is a $g$-clause of $C$. Now, we may infer that $\{D'', l l_2 \bot \approx r l_2 \bot\} \models D'$, where $D'' := C' \vee l_1 r l_2 \bot \not\approx v\bot$ is a $g$-clause of $C \vee l_1 r l_2 \not\approx v$ and $l l_2 \bot \approx r l_2 \bot$ is a $g$-equation of $l \approx r$. We also have $D' \succ_g D''$ and $D' \succ_g l l_2 \bot \approx r l_2 \bot$, and thus the conclusion follows.

Otherwise, suppose that $l \approx r$ simplifies $D := C \vee l_1 l l_2 \approx v$ into $C \vee l_1 r l_2 \approx v$. Then every $g$-clause $D'$ of $D$ has the form $D' := C' \vee l_1 l l_2 w \bot \approx v w \bot$ for some $w \in \Sigma^*$, where $C'$ is a $g$-clause of $C$. Now, we have $\{D'', l l_2 w \bot \approx r l_2 w \bot\} \models D'$, where $D'' := C' \vee l_1 r l_2 w \bot \approx v w \bot$ is a $g$-clause of $C \vee l_1 r l_2 \approx v$ for some $w \in \Sigma^*$ and $l l_2 w \bot \approx r l_2 w \bot$ is a $g$-equation of $l \approx r$. We also have $D' \succ_g D''$ and $D' \succ_g l l_2 w \bot \approx r l_2 w \bot$ because $l_1$ is not $\lambda$ in the condition of the rule (i.e., $l_1 l l_2 w \bot \succ_g l l_2 w \bot$), and thus the conclusion follows. □

We see that if $C$ subsumes $C'$ with $C$ and $C'$ containing the same number of literals, then they are the same when viewed as the finite multisets, so we can remove $C'$. Therefore, we exclude this case in the following lemma.

**Lemma 6.** *If a clause $C$ subsumes a clause $D$ and $C$ contains fewer literals than $D$, then $D$ is redundant w.r.t. $\{C\}$.*

*Proof.* Suppose that $C$ subsumes $D$ and $C$ contains fewer literals than $D$. Then $D$ can be denoted by $C \vee B$ for some nonempty clause $B$. Now, for every $g$-clause $D' := C' \vee B'$ of $D$, we have $\{C'\} \models D'$ with $D' \succ_g C'$, where $C'$ and $B'$ are $g$-clauses of $C$ and $B$, respectively. Thus, $D$ is redundant w.r.t. $\{C\}$. □

**Lemma 7.** *A tautology $C \vee s \approx s$ is redundant.*

*Proof.* It is easy to see that for every $g$-clause $C' \vee su\bot \approx su\bot$ of $C \vee s \approx s$, we have $\models C' \vee su\bot \approx su\bot$, where $u \in \Sigma^*$ and $C'$ is a $g$-clause of $C$. Thus, $C \vee s \approx s$ is redundant. □

## 5 Refutational Completeness

In this section, we adapt the model construction and equational theorem proving techniques used in [3, 18, 22] and show that $\mathfrak{S}$ with the contraction rules is refutationally complete.

**Definition 8.** A $g$-equation $s\bot \approx t\bot$ is *reductive* for a $g$-clause $C := D \vee s\bot \approx t\bot$ if $s\bot \approx t\bot$ is strictly maximal (w.r.t. $\succ_g$) in $C$ with $s\bot \succ_g t\bot$.

**Definition 9.** (Model Construction) Let $S$ be a set of clauses over $\Sigma^*$. We use induction on $\succ_g$ to define the sets $R_C, E_C$, and $I_C$ for all $g$-clauses $C$ of clauses in $S$. Let $C$ be such a $g$-clause of a clause in $S$ and suppose that $E_{C'}$ has been defined for all $g$-clauses $C'$ of clauses in $S$ for which $C \succ_g C'$. Then we define by $R_C = \bigcup_{C \succ_g C'} E_{C'}$. We also define by $I_C$ the equality interpretation $R_C^*$, which denotes the least congruence containing $R_C$.

Now, let $C := D \vee s\bot \approx t\bot$ such that $C$ is not a $g$-clause of a clause with a selected literal in $S$. Then $C$ produces $E_C = \{s\bot \to t\bot\}$ if the following conditions are met: (1) $I_C \not\models C$, (2) $I_C \not\models t\bot \approx t'\bot$ for every $s\bot \approx t'\bot$ in $D$, (3) $s\bot \approx t\bot$ is reductive for $C$, and (4) $s\bot$ is irreducible by $R_C$. We say that $C$ is *productive* and produces $E_C$ if it satisfies all of the above conditions. Otherwise, $E_C = \emptyset$. Finally, we define $I_S$ as the equality interpretation $R_S^*$, where $R_S = \bigcup_C E_C$ is the set of all $g$-rules produced by $g$-clauses of clauses in $S$.

**Lemma 10.** *(i) $R_S$ has the Church-Rosser property.*
*(ii) $R_S$ is terminating.*
*(iii) For $g$-terms $u\bot$ and $v\bot$, $I_S \models u\bot \approx v\bot$ if and only if $u\bot \downarrow_{R_S} v\bot$.*
*(iv) If $I_S \models s \approx t$, then $I_S \models usv \approx utv$ for nonempty strings $s,t,u,v \in \Sigma^*$.*

*Proof.* (i) $R_S$ is left-reduced because there are no overlaps among the left-hand sides of rewrite rules in $R_S$, and thus $R_S$ has the Church-Rosser property.
(ii) For each rewrite rule $l\bot \to r\bot$ in $R_S$, we have $l\bot \succ_g r\bot$, and thus $R_S$ is terminating.
(iii) Since $R_S$ has the Church-Rosser property and is terminating by (i) and (ii), respectively, $R_S$ is convergent. Thus, $I_S \models u\bot \approx v\bot$ if and only if $u\bot \downarrow_{R_S} v\bot$ for $g$-terms $u\bot$ and $v\bot$.
(iv) Suppose that $I_S \models s \approx t$ for nonempty strings $s$ and $t$. Then, we have $I_S \models svw\bot \approx tvw\bot$ for all strings $v$ and $w$ by Definition 3(i). Similarly, since $I_S$ is an equality Herbrand interpretation, we also have $I_S \models usvw\bot \approx utvw\bot$ for all strings $u$, which means that $I_S \models usv \approx utv$ by Definition 3(i). □

Lemma 10(iv) says that the monotonicity assumption used in this paper holds w.r.t. a model constructed by Definition 9.

**Definition 11.** Let $S$ be a set of clauses over $\Sigma^*$. We say that $S$ is *saturated* under $\mathfrak{S}$ if every inference by $\mathfrak{S}$ with premises in $S$ is redundant w.r.t. $S$.

**Definition 12.** Let $C := s_1 \approx t_1 \vee \cdots \vee s_m \approx t_m \vee u_1 \not\approx v_1 \vee \cdots \vee u_n \not\approx v_n$ be a clause over $\Sigma^*$, and $C' = s_1w_1\bot \approx t_1w_1\bot \vee \cdots \vee s_mw_m\bot \approx t_mw_m\bot \vee u_1\bot \not\approx v_1\bot \vee \cdots \vee u_n\bot \not\approx v_n\bot$ for some strings $w_1, \ldots, w_m$ be a $g$-clause of $C$. We say that $C'$ is a *reduced $g$-clause* of $C$ w.r.t. a rewrite system $R$ if every $w_i\bot$, $1 \le i \le m$, is not reducible by $R$.

In the proof of the following lemma, we write $s[t]_{suf}$ to indicate that $t$ occurs in $s$ as a suffix and (ambiguously) denote by $s[u]_{suf}$ the result of replacing the occurrence of $t$ (as a suffix of $s$) by $u$.

**Lemma 13.** *Let $S$ be saturated under $\mathfrak{S}$ not containing the empty clause and $C$ be a $g$-clause of a clause in $S$. Then $C$ is true in $I_S$. More specifically,*
*(i) If $C$ is redundant w.r.t. $S$, then it is true in $I_S$.*

*(ii) If $C$ is not a reduced $g$-clause of a clause in $S$ w.r.t. $R_S$, then it is true in $I_S$.*
*(iii) If $C := C' \vee s\bot \approx t\bot$ produces the rule $s\bot \to t\bot$, then $C'$ is false and $C$ is true in $I_S$.*
*(iv) If $C$ is a $g$-clause of a clause in $S$ with a selected literal, then it is true in $I_S$.*
*(v) If $C$ is non-productive, then it is true in $I_S$.*

*Proof.* We use induction on $\succ_g$ and assume that (i)–(v) hold for every $g$-clause $D$ of a clause in $S$ with $C \succ_g D$.

(i) Suppose that $C$ is redundant w.r.t. $S$. Then there exist $g$-clauses $C'_1, \ldots, C'_k$ of clauses $C_1, \ldots, C_k$ in $S$, such that $\{C'_1, \ldots, C'_k\} \models C$ and $C \succ_g C'_i$ for all $1 \leq i \leq k$. By the induction hypothesis, each $C'_i$, $1 \leq i \leq k$, is true in $I_S$. Thus, $C$ is true in $I_S$.

(ii) Suppose that $C$ is a $g$-clause of a clause $B := s_1 \approx t_1 \vee \cdots \vee s_m \approx t_m \vee u_1 \not\approx v_1 \vee \cdots \vee u_n \not\approx v_n$ in $S$ but is not a reduced $g$-clause w.r.t. $R_S$. Then $C$ is of the form $C := s_1 w_1\bot \approx t_1 w_1\bot \vee \cdots \vee s_m w_m\bot \approx t_m w_m\bot \vee u_1\bot \not\approx v_1\bot \vee \cdots \vee u_n\bot \not\approx v_n\bot$ for $w_1, \ldots, w_m \in \Sigma^*$ and some $w_k\bot$ is reducible by $R_S$. Now, consider $C' = s_1 w'_1\bot \approx t_1 w'_1\bot \vee \cdots \vee s_m w'_m\bot \approx t_m w'_m\bot \vee u_1\bot \not\approx v_1\bot \vee \cdots \vee u_n\bot \not\approx v_n\bot$, where $w'_i\bot$ is the normal form of $w_i\bot$ w.r.t. $R_S$ for each $1 \leq i \leq m$. Then $C'$ is a reduced $g$-clause of $B$ w.r.t. $R_S$, and is true in $I_S$ by the induction hypothesis. Since each $w_i\bot \approx w'_i\bot$, $1 \leq i \leq m$, is true in $I_S$ by Lemma 10(iii), we may infer that $C$ is true in $I_S$.

In the remainder of the proof of this lemma, we assume that $C$ is neither redundant w.r.t. $S$ nor is it a reducible $g$-clause w.r.t. $R_S$ of some clause in $S$. (Otherwise, we are done by (i) or (ii).)

(iii) Suppose that $C := C' \vee s\bot \approx t\bot$ produces the rule $s\bot \to t\bot$. Since $s\bot \to t\bot \in E_C \subset R_S$, we see that $C$ is true in $I_S$. We show that $C'$ is false in $I_S$. Let $C' := \Gamma \to \Delta$. Then $I_C \not\models C'$ by Definition 9, which implies that $I_C \cap \Delta = \emptyset, I_C \supseteq \Gamma$, and thus $I_S \supseteq \Gamma$. It remains to show that $I_S \cap \Delta = \emptyset$. Suppose to the contrary that $\Delta$ contains an equation $s'\bot \approx t'\bot$ which is true in $I_S$. Since $I_C \cap \Delta = \emptyset$, we must have $s'\bot \approx t'\bot \in I \setminus I_C$, which is only possible if $s\bot = s'\bot$ and $I_C \models t\bot \approx t'\bot$, contradicting condition (2) in Definition 9.

(iv) Suppose that $C$ is of the form $C := B' \vee s\bot \not\approx t\bot$, where $s\bot \not\approx t\bot$ is a $g$-literal of a selected literal in a clause in $S$ and $B'$ is a $g$-clause of $B$.

(iv.1) If $s\bot = t\bot$, then $B'$ is an equality resolvent of $C$ and the Equality Resolution inferences can be lifted. By saturation of $S$ under $\mathfrak{S}$ and the induction hypothesis, $B'$ is true in $I_S$. Thus, $C$ is true in $I_S$.

(iv.2) If $s\bot \neq t\bot$, then suppose to the contrary that $C$ is false in $I_S$. Then we have $I_S \models s\bot \approx t\bot$, which implies that $s\bot$ or $t\bot$ is reducible by $R_S$ by Lemma 10(iii). Without loss of generality, we assume that $s\bot$ is reducible by $R_S$ with some rule $lu\bot \to ru\bot$ for some $u \in \Sigma^*$ produced by a productive $g$-clause $D' \vee lu\bot \approx ru\bot$ of a clause $D \vee l \approx r \in S$. This means that $s\bot$ has a suffix $lu\bot$. Now, consider the following inference by Rewriting:

$$\frac{B \vee s[lu]_{suf} \not\approx t \qquad D \vee l \approx r}{B \vee D \vee s[ru]_{suf} \not\approx t}$$

where $s[lu]_{suf} \not\approx t$ is selected for the left premise. The conclusion of the above inference has a $g$-clause $C' := B' \vee D' \vee s\bot[ru\bot]_{suf} \not\approx t\bot$. By saturation of $S$ under $\mathfrak{S}$ and the induction hypothesis, $C'$ must be true in $I_S$. Moreover, we see that $s\bot[ru\bot]_{suf} \not\approx t\bot$ is false in $I_S$ by Lemma 10(iii), and $D'$ are false in $I_S$ by (iii). This means that $B'$ is true in $I_S$, and thus $C$ (i.e., $C = B' \vee s\bot \not\approx t\bot$) is true in $I_S$, which is the required contradiction.

(v) If $C$ is non-productive, then we assume that $C$ is not a $g$-clause of a clause with a selected literal. Otherwise, the proof is done by (iv). This means that $C$ is of the form $C := B' \vee su\bot \approx tu\bot$, where $su\bot \approx tu\bot$ is maximal in $C$ and $B'$ contains no selected literal. If $su\bot = tu\bot$, then we are done. Therefore, without loss of generality, we assume that $su\bot \succ_g tu\bot$. As $C$ is non-productive, it means that (at least)

one of the conditions in Definition 9 does not hold.

If condition (1) does not hold, then $I_C \models C$, so we have $I_S \models C$, i.e., $C$ is true in $I_S$. If condition (1) holds but condition (2) does not hold, then $C$ is of the form $C := B'_1 \vee su\bot \approx tu\bot \vee svw\bot \approx t'vw\bot$, where $su = svw$ (i.e., $u = vw$) and $I_C \models tu\bot \approx t'vw\bot$.

Suppose first that $tu\bot = t'vw\bot$. Then we have $t = t'$ since $u = vw$. Now, consider the following inference by Factoring:

$$\frac{B_1 \vee s \approx t \vee sv \approx tv}{B_1 \vee sv \approx tv}$$

The conclusion of the above inference has a $g$-clause $C' := B'_1 \vee svw\bot \approx tvw\bot$, i.e., $C' := B'_1 \vee su\bot \approx tu\bot$ since $u = vw$. By saturation of $S$ under $\mathfrak{S}$ and the induction hypothesis, $C'$ is true in $I_S$, and thus $C$ is true in $I_S$.

Otherwise, suppose that $tu\bot \neq t'vw\bot$. Then we have $tu\bot \downarrow_{R_C} t'vw\bot$ by Lemma 10(iii) and $tu\bot \succ_g t'vw\bot$ because $su\bot \approx tu\bot$ is maximal in $C$. This means that $tu\bot$ is reducible by $R_C$ by some rule $l\tau\bot \to r\tau\bot$ produced by a productive $g$-clause $D' \vee l\tau\bot \approx r\tau\bot$ of a clause $D \vee l \approx r \in S$. Now, we need to consider two cases:

(v.1) If $t$ has the form $t := u_1u_2$ and $l$ has the form $l := u_2u_3$, then consider the following inference by Paramodulation:

$$\frac{B \vee s \approx u_1u_2 \qquad D \vee u_2u_3 \approx r}{B \vee D \vee su_3 \approx u_1r}$$

The conclusion of the above inference has a $g$-clause $C' := B' \vee D' \vee su_3\tau\bot \approx u_1r\tau\bot$ with $u = u_3\tau$. By saturation of $S$ under $\mathfrak{S}$ and the induction hypothesis, $C'$ is true in $I_S$. Since $D'$ is false in $I_S$ by (iii), either $B'$ or $su_3\tau\bot \approx u_1r\tau\bot$ is true in $I_S$. If $B'$ is true in $I_S$, so is $C$. If $su_3\tau\bot \approx u_1r\tau\bot$ is true in $I_S$, then $su\bot \approx tu\bot$ is also true in $I_S$ by Lemma 10(iii), where $t = u_1u_2$ and $u = u_3\tau$. Thus, $C$ is true in $I_S$.

(v.2) If $t$ has the form $t := u_1u_2u_3$ and $l$ has the form $l := u_2$, then consider the following inference by Rewrite:

$$\frac{B \vee s \approx u_1u_2u_3 \qquad D \vee u_2 \approx r}{B \vee D \vee s \approx u_1ru_3}$$

The conclusion of the above inference has a $g$-clause $C'' := B' \vee D' \vee su\bot \approx u_1ru_3u\bot$ with $\tau = u_3u$. By saturation of $S$ under $\mathfrak{S}$ and the induction hypothesis, $C''$ is true in $I_S$. Since $D'$ is false in $I_S$ by (iii), either $B'$ or $su\bot \approx u_1ru_3u\bot$ is true in $I_S$. Similarly to case (v.1), if $B'$ is true in $I_S$, so is $C$. If $su\bot \approx u_1ru_3u\bot$ is true in $I_S$, then $su\bot \approx tu\bot$ is also true in $I_S$ by Lemma 10(iii), where $t = u_1u_2u_3$. Thus, $C$ is true in $I_S$.

If conditions (1) and (2) hold but condition (3) does not hold, then $su\bot \approx tu\bot$ is only maximal but is not strictly maximal, so we are in the previous case. (Since $\succ_g$ is total on $g$-clauses, condition (2) does not hold.) If conditions (1)–(3) hold but condition (4) does not hold, then $su\bot$ is reducible by $R_C$ by some rule $l\tau\bot \to r\tau\bot$ produced by a productive $g$ clause $D' \vee l\tau\bot \approx r\tau\bot$ of a clause $D \vee l \approx r \in S$. Again, we need to consider two cases:

(v.1') If $s$ has the form $s := u_1u_2$ and $l$ has the form $l := u_2u_3$, then consider the following inference by Superposition:

$$\frac{B \vee u_1u_2 \approx t \qquad D \vee u_2u_3 \approx r}{B \vee D \vee u_1r \approx tu_3}$$

The conclusion of the above inference has a $g$-clause $C' := B' \vee D' \vee u_1 r\tau\bot \approx tu_3\tau\bot$ with $u = u_3\tau$. By saturation of $S$ under $\mathfrak{S}$ and the induction hypothesis, $C'$ is true in $I_S$. Since $D'$ is false in $I_S$ by (iii), either $B'$ or $u_1 r\tau\bot \approx tu_3\tau\bot$ is true in $I_S$. If $B'$ is true in $I_S$, so is $C$. If $u_1 r\tau\bot \approx tu_3\tau\bot$ is true in $I_S$, then $su\bot \approx tu\bot$ is also true in $I_S$ by Lemma 10(iii), where $s = u_1u_2$ and $u = u_3\tau$. Thus, $C$ is true in $I_S$.

(v.2') If $s$ has the form $s := u_1u_2u_3$ and $l$ has the form $l := u_2$, then consider the following inference by Rewrite:

$$\frac{B \vee u_1u_2u_3 \approx t \qquad D \vee u_2 \approx r}{B \vee D \vee u_1ru_3 \approx t}$$

The conclusion of the above inference has a $g$-clause $C'' := B' \vee D' \vee u_1ru_3u\bot \approx tu\bot$ with $\tau = u_3u$. By saturation of $S$ under $\mathfrak{S}$ and the induction hypothesis, $C''$ is true in $I_S$. Since $D'$ is false in $I_S$ by (iii), either $B'$ or $u_1ru_3u\bot \approx tu\bot$ is true in $I_S$. Similarly to case (v.1'), If $B'$ is true in $I_S$, so is $C$. If $u_1ru_3u\bot \approx tu\bot$ is true in $I_S$, then $su\bot \approx tu\bot$ is also true in $I_S$ by Lemma 10(iii), where $s = u_1u_2u_3$. Thus, $C$ is true in $I_S$. □

**Definition 14.** (i) A *theorem proving derivation* is a sequence of sets of clauses $S_0 = S, S_1, \ldots$ over $\Sigma^*$ such that:

(i.1) Deduction: $S_i = S_{i-1} \cup \{C\}$ if $C$ can be deduced from premises in $S_{i-1}$ by applying an inference rule in $\mathfrak{S}$.

(i.2) Deletion: $S_i = S_{i-1} \setminus \{D\}$ if $D$ is redundant w.r.t. $S_{i-1}$.[5]

(ii) The set $S_\infty := \bigcup_i(\bigcap_{j\geq i} S_j)$ is the *limit* of the theorem proving derivation.

We see that the soundness of a theorem proving derivation w.r.t. the proposed inference system is straightforward, i.e., $S_i \models S_{i+1}$ for all $i \geq 0$.

**Definition 15.** A theorem proving derivation $S_0, S_1, S_2, \ldots$ is *fair* w.r.t. the inference system $\mathfrak{S}$ if every inference by $\mathfrak{S}$ with premises in $S_\infty$ is redundant w.r.t. $\bigcup_j S_j$.

**Lemma 16.** *Let $S$ and $S'$ be sets of clauses over $\Sigma^*$.*
*(i) If $S \subseteq S'$, then any clause which is redundant w.r.t. $S$ is also redundant w.r.t. $S'$.*
*(ii) If $S \subseteq S'$ and all clauses in $S' \setminus S$ are redundant w.r.t. $S'$, then any clause or inference which is redundant w.r.t. $S'$ is also redundant w.r.t. $S$.*

*Proof.* The proof of part (i) is obvious. For part (ii), suppose that a clause $C$ is redundant w.r.t. $S'$ and let $C'$ be a $g$-clause of it. Then there exists a minimal set $N := \{C'_1, \ldots, C'_n\}$ (w.r.t. $\succ_g$) of $g$-clauses of clauses in $S'$ such that $N \models C'$ and $C' \succ_g C'_i$ for all $1 \leq i \leq n$. We claim that all $C'_i$ in $N$ are not redundant w.r.t. $S'$, which shows that $C'$ is redundant w.r.t. $S$. Suppose to the contrary that some $C'_j$ is redundant w.r.t. $S'$. Then there exist a set $N' := \{D'_1, \ldots, D'_m\}$ of $g$-clauses of clauses in $S'$ such that $N' \models C'_j$ and $C'_j \succ_g D'_i$ for all $1 \leq i \leq m$. This means that we have $\{C'_1, \ldots, C'_{j-1}, D'_1, \ldots, D'_m, C'_{j+1}, \ldots, C'_n\} \models C'$, which contradicts our minimal choice of the set $N = \{C'_1, \ldots, C'_n\}$.

Next, suppose an inference $\pi$ with conclusion $D$ is redundant w.r.t. $S'$ and let $\pi'$ be a $g$-instance of it such that $B$ is the maximal premise and $D'$ is the conclusion of $\pi'$ (i.e., a $g$-clause of $D$). Then there exists a minimal set $P := \{D'_1, \ldots, D'_n\}$ (w.r.t. $\succ_g$) of $g$-clauses of clauses in $S'$ such that $P \models D'$ and $B \succ_g D'_i$ for all $1 \leq i \leq n$. As above, we may infer that all $D'_i$ in $P$ are not redundant w.r.t. $S'$, and thus $\pi'$ is redundant w.r.t. $S$. □

**Lemma 17.** *Let $S_0, S_1, \ldots$ be a fair theorem proving derivation w.r.t. $\mathfrak{S}$. Then $S_\infty$ is saturated under $\mathfrak{S}$.*

[5] Here, an inference by Simplification combines the Deduction step for $C \vee l_1rl_2 \bowtie v$ and the Deletion step for $C \vee l_1ll_2 \bowtie v$ (see the Simplification rule).

*Proof.* If $S_\infty$ contains the empty clause, then it is obvious that $S_\infty$ is saturated under $\mathfrak{S}$. Therefore, we assume that the empty clause is not in $S_\infty$.

If a clause $C$ is deleted in a theorem proving derivation, then $C$ is redundant w.r.t. some $S_j$. By Lemma 16(i), it is also redundant w.r.t. $\bigcup_j S_j$. Similarly, every clause in $\bigcup_j S_j \setminus S_\infty$ is redundant w.r.t. $\bigcup_j S_j$.

By fairness, every inference $\pi$ by $\mathfrak{S}$ with premises in $S_\infty$ is redundant w.r.t. $\bigcup_j S_j$. Using Lemma 16(ii) and the above, $\pi$ is also redundant w.r.t. $S_\infty$, which means that $S_\infty$ is saturated under $\mathfrak{S}$. □

**Theorem 18.** *Let $S_0, S_1, \ldots$ be a fair theorem proving derivation w.r.t. $\mathfrak{S}$. If $S_\infty$ does not contain the empty clause, then $I_{S_\infty} \models S_0$ (i.e., $S_0$ is satisfiable.)*

*Proof.* Suppose that $S_0, S_1, \ldots$ is a fair theorem proving derivation w.r.t. $\mathfrak{S}$ and that its limit $S_\infty$ does not contain the empty clause. Then $S_\infty$ is saturated under $\mathfrak{S}$ by Lemma 17. Let $C'$ be a $g$-clause of a clause $C$ in $S_0$. If $C \in S_\infty$, then $C'$ is true in $I_{S_\infty}$ by Lemma 13. Otherwise, if $C \notin S_\infty$, then $C$ is redundant w.r.t. some $S_j$. It follows that $C$ redundant w.r.t. $\bigcup_j S_j$ by Lemma 16(i), and thus redundant w.r.t. $S_\infty$ by Lemma 16(ii). This means that there exist $g$-clauses $C'_1, \ldots, C'_k$ of clauses $C_1, \ldots, C_k$ in $S_\infty$ such that $\{C'_1, \ldots, C'_k\} \models C'$ and $C' \succ_g C'_i$ for all $1 \leq i \leq k$. Since each $C'_i$, $1 \leq i \leq k$, is true in $I_{S_\infty}$ by Lemma 13, $C'$ is also true in $I_{S_\infty}$, and thus the conclusion follows. □

The following theorem states that $\mathfrak{S}$ with the contraction rules is refutationally complete for clauses over $\Sigma^*$.

**Theorem 19.** *Let $S_0, S_1, \ldots$ be a fair theorem proving derivation w.r.t. $\mathfrak{S}$. Then $S_0$ is unsatisfiable if and only if the empty clause is in some $S_j$.*

*Proof.* Suppose that $S_0, S_1, \ldots$ be a fair theorem proving derivation w.r.t. $\mathfrak{S}$. By the soundness of the derivation, if the empty clause is in some $S_j$, then $S_0$ is unsatisfiable. Otherwise, if the empty clause is not in $S_k$ for all $k$, then $S_\infty$ does not contain the empty clause by the soundness of the derivation. Applying Theorem 18, we conclude that $S_0$ is satisfiable. □

## 6 Conditional Completion

In this section, we present a saturation procedure under $\mathfrak{S}$ for a set of conditional equations over $\Sigma^*$, where a conditional equation is naturally written as an equational Horn clause. A saturation procedure under $\mathfrak{S}$ can be viewed as *conditional completion* [12] for a set of conditional equations over $\Sigma^*$. If a set of conditional equations over $\Sigma^*$ is simply a set of equations over $\Sigma^*$, then the proposed saturation procedure (w.r.t. $\succ$) corresponds to a completion procedure for a string rewriting system. Conditional string rewriting systems were considered in [11] in the context of embedding a finitely generated monoid with decidable word problem into a monoid presented by a finite convergent conditional presentation. It neither discusses a conditional completion (or a saturation) procedure, nor considers the word problems for conditional equations over $\Sigma^*$ in general.

First, it is easy to see that a set of equations over $\Sigma^*$ is consistent. Similarly, a set of conditional equations $R$ over $\Sigma^*$ is consistent because each conditional equation has always a positive literal and we cannot derive the empty clause from $R$ using a saturation procedure under $\mathfrak{S}$ that is refutationally complete (cf. Section 9 in [13]). Since we only consider Horn clauses in this section, we neither need to consider the Factoring rule nor the Paramodulation rule in $\mathfrak{S}$. In the remainder of this section, by a conditional equational theory $R$, we mean a set of conditional equations $R$ over $\Sigma^*$.

**Definition 20.** Given a conditional equational theory $R$ and two finite words $s,t \in \Sigma^*$, a *word problem* w.r.t. $R$ is of the form $\phi := s \approx^? t$. The *goal* of this word problem is $s \not\approx t$. We say that a word problem $s \approx^? t$ w.r.t. $R$ is *decidable* if there is a decision procedure for determining whether $s \approx t$ is entailed by $R$ (i.e., $R \models s \approx t$) or not (i.e., $R \not\models s \approx t$) .

Given a conditional equational theory $R$, let $G := s \not\approx t$ be the goal of a word problem $s \approx^? t$ w.r.t. $R$. (Note that $G$ does not have any positive literal.) Then we see that $R \cup \{s \approx t\}$ is consistent if and only if $R \cup \{G\}$ is inconsistent. This allows one to decide a word problem w.r.t. $R$ using the equational theorem proving procedure discussed in Section 5.

**Lemma 21.** *Let $R$ be a conditional equational theory finitely saturated under $\mathfrak{S}$. Then Rewrite together with Equality Resolution is terminating and refutationally complete for $R \cup \{G\}$, where $G$ is the goal of a word problem w.r.t. $R$.*

*Proof.* Since $R$ is already saturated under $\mathfrak{S}$, inferences among Horn clauses in $R$ are redundant and remain redundant in $R \cup \{G\}$ for a theorem proving derivation starting with $R \cup \{G\}$. (Here, $\{G\}$ can be viewed as a *set of support* [3] for a refutation of $R \cup \{G\}$.) Now, observe that $G$ is a negative literal, so it should be selected. The only inference rules in $\mathfrak{S}$ involving a selected literal are the Rewrite and Equality Resolution rule. Furthermore, the derived literals from $G$ w.r.t. Rewrite will also be selected eventually. Therefore, it suffices to consider positive literals as the right premise (because they contain no selected literal), and $G$ and its derived literals w.r.t. Rewrite as the left premise for the Rewrite rule. Observe also that if $G'$ is an immediate derived literal from $G$ w.r.t. Rewrite, then we see that $G \succ G'$. If $G$ or its derived literal from $G$ w.r.t. Rewrite becomes of the form $u \not\approx u$ for some $u \in \Sigma^*$, then it will also be selected and an Equality Resolution inference yields the empty clause. Since $\succ$ is terminating and there are only finitely many positive literals in $R$, we may infer that the Rewrite and Equality Resolution inference steps on $G$ and its derived literals are terminating. (The number of positive literals in $R$ remains the same during a theorem proving derivation starting with $R \cup \{G\}$ using our selection strategy.)

Finally, since $\mathfrak{S}$ is refutationally complete by Thereom 19, Rewrite together with Equality Resolution is also refutationally complete for $R \cup \{G\}$. □

Given a finitely saturated conditional equational theory $R$ under $\mathfrak{S}$, we provide a decision procedure for the word problems w.r.t. $R$ in the following theorem.

**Theorem 22.** *Let $R$ be a conditional equational theory finitely saturated under $\mathfrak{S}$. Then the word problems w.r.t. $R$ are decidable by Rewrite together with Equality Resolution.*

*Proof.* Let $\phi := s \approx^? t$ be a word problem w.r.t. $R$ and $G$ be the goal of $\phi$. We know that by Lemma 21, Rewrite together with Equality Resolution is terminating and refutationally complete for $R \cup \{G\}$. Let $R_0 := R \cup \{G\}, R_1, \ldots, R_n$ be a fair theorem proving derivation w.r.t. Rewrite together with Equality Resolution such that $R_n$ is the limit of this derivation. If $R_n$ contains the empty clause, then $R_n$ is inconsistent, and thus $R_0$ is inconsistent, i.e., $\{s \not\approx t\} \cup R$ is inconsistent by the soundness of the derivation. Since $R$ is consistent and $\{s \not\approx t\} \cup R$ is saturated under $\mathfrak{S}$, we may infer that $R \models s \approx t$.

Otherwise, if $R_n$ does not contain the empty clause, then $R_n$ is consistent, and thus $R_0$ is consistent by Theorem 19, i.e., $\{s \not\approx t\} \cup R$ is consistent. Since $R$ is consistent and $\{s \not\approx t\} \cup R$ is saturated under $\mathfrak{S}$, we may infer that $R \not\models s \approx t$. □

The following corollary is a consequence of Theorem 22 and the following observation. Let $R = R_0, R_1, \ldots, R_n$ be a finite fair theorem proving derivation w.r.t. $\mathfrak{S}$ for an initial conditional equational theory $R$ with the limit $\bar{R} := R_n$. Then $R \cup \{G\}$ is inconsistent if and only if $\bar{R} \cup \{G\}$ is inconsistent by the soundness of the derivation and Theorem 19.

**Corollary 23.** *Let $R = R_0, R_1, \ldots$ be a fair theorem proving derivation w.r.t. $\mathfrak{S}$ for a conditional equational theory $R$. If $R$ can be finitely saturated under $\mathfrak{S}$, then the word problems w.r.t. $R$ are decidable.*

**Example 5.** Let $a \succ b \succ c$ and $R$ be a conditional equational theory consisting of the following rules 1: $aa \approx \lambda$, 2: $bb \approx \lambda$, 3: $ab \approx \lambda$, 4: $ab \not\approx ba \vee ac \approx ca$, and 5: $ab \not\approx ba \vee ac \not\approx ca \vee bc \approx cb$. We first saturate $R$ under $\mathfrak{S}$:

6: $\lambda \not\approx ba \vee ac \approx ca$ ($ab \not\approx ba$ is selected for 4. Rewrite of 4 with 3)
7: $\lambda \not\approx ba \vee ac \not\approx ca \vee bc \approx cb$ ($ab \not\approx ba$ is selected for 5. Rewrite of 5 with 3)
8: $a \approx b$ (Superposition of 1 with 3)
9: $\lambda \not\approx bb \vee ac \approx ca$ ($\lambda \not\approx ba$ is selected for 6. Rewrite of 6 with 8)
10: $\lambda \not\approx \lambda \vee ac \approx ca$ ($\lambda \not\approx bb$ is selected for 9. Rewrite of 9 with 2)
11: $ac \approx ca$ ($\lambda \not\approx \lambda$ is selected for 10. Equality Resolution on 10)
12: $\lambda \not\approx bb \vee ac \not\approx ca \vee bc \approx cb$ ($\lambda \not\approx ba$ is selected for 7. Rewrite of 7 with 8)
13: $\lambda \not\approx \lambda \vee ac \not\approx ca \vee bc \approx cb$ ($\lambda \not\approx bb$ is selected for 12. Rewrite of 12 with 2)
14: $ac \not\approx ca \vee bc \approx cb$ ( $\lambda \not\approx \lambda$ is selected for 13. Equality Resolution on 13)
15: $ca \not\approx ca \vee bc \approx cb$ ( $ac \not\approx ca$ is selected for 14. Rewrite of 14 with 11)
16: $bc \approx cb$ ($ca \not\approx ca$ is selected for 15. Equality Resolution on 15)
...
After some simplification steps, we have a saturated set $\bar{R}$ for $R$ under $\mathfrak{S}$ using our selection strategy (i.e., the selection of negative literals). We may infer that the positive literals in $\bar{R}$ are as follows. $1' : bb \approx \lambda$, $2' : a \approx b$, and $3' : bc \approx cb$. Note that only the positive literals in $\bar{R}$ are now needed to solve a word problem w.r.t. $R$ because of our selection strategy.

Now, consider the word problem $\phi := acbcba \approx^? bccaba$ w.r.t. $R$, where the goal of $\phi$ is $G := acbcba \not\approx bccaba$. We only need the Rewrite and Equality Resolution steps on $G$ and its derived literals from $G$ using $1'$, $2'$, and $3'$. Note that all the following literals are selected except the empty clause.

$4'$: $bcbcbb \not\approx bccbbb$ (Rewrite steps of $G$ and its derived literals from $G$ using $2'$).
$5'$: $bcbc \not\approx bccb$ (Rewrite steps of $4'$ and its derived literals from $4'$ using $1'$).
$6'$: $ccbb \not\approx ccbb$ (Rewrite steps of $5'$ and its derived literals from $5'$ using $3'$).
$7'$: $\square$ (Equality Resolution on $6'$)

Since $\bar{R} \cup G$ is inconsistent, we see that $R \cup G$ is inconsistent by the soundness of the derivation, where $R$ and $\bar{R}$ are consistent. Therefore, we may infer that $R \models acbcba \approx bccaba$.

# 7 Related Work

Equational reasoning on strings has been studied extensively in the context of string rewriting systems and Thue systems [8] and their related algebraic structures. The monotonicity assumption used in this paper is found in string rewriting systems and Thue systems in the form of a congruence relation (see [8, 17]). See [7, 10, 21, 23] also for the completion of algebraic structures and decidability results using string rewriting systems. However, those systems are not concerned with equational theorem proving for general clauses over strings. If the monotonicity assumption is discarded, then equational theorem proving for clauses over strings can be handled by traditional superposition calculi or SMT with the theory of equality with uninterpreted functions (EUF) and their variants [6] using a simple translation into first-order ground terms. Also, efficient SMT solvers for various string constraints were discussed

in the literature (e.g., [20]).

Meanwhile, equational theorem proving modulo associativity was studied in [26]. (See also [19] for equational theorem proving with *sequence variables* and fixed or variadic arity symbols). This approach is not tailored towards (ground) strings, so we need an additional encoding for each string. Also, it is not efficient and does not provide a similar decision procedure discussed in Section 6.

The proposed calculus is the first sound and refutationally complete equational theorem proving calculus for general clauses over strings under the monotonicity assumption. One may attempt to use the existing superposition calculi for clauses over strings with the proposed translation scheme, which translates clauses over strings into clauses over first-order terms discussed in Section 3.2. However, this does not work because of the Equality Factoring rule [3, 22] or the Merging Paramodulation rule [3], which is essential for first-order superposition theorem proving calculi in general. For example, consider a clause $a \approx b \vee a \approx c$ with $a \succ b \succ c$, which is translated into a first-order clause $a(x) \approx b(x) \vee a(y) \approx c(y)$. The Equality Factoring rule yields $b(z) \not\approx c(z) \vee a(z) \approx c(z)$ from $a(x) \approx b(x) \vee a(y) \approx c(y)$, which cannot be translated back into a clause over strings (see Lemma 1). Similarly, a first-order clause produced by Merging Paramodulation may not be translated back into a clause over strings. If one is only concerned with refutational completeness, then the existing superposition calculi[6] can be adapted by using the proposed translation scheme. In this case, a saturated set may not be translated back into clauses over strings in some cases, which is an obvious drawback for its applications (see *programs* in [3]).

# 8 Conclusion

This paper has presented a new refutationally complete superposition calculus with strings and provided a framework for equational theorem proving for clauses over strings. The results presented in this paper generalize the results about completion of string rewriting systems and equational theorem proving using equations over strings. The proposed superposition calculus is based on the simple string matching methods and the efficient length-lexicographic ordering that allows one to compare two finite strings in linear time for a fixed signature with its precedence.

The proposed approach translates for a clause over strings into the first-order representation of the clause by taking the monotonicity property of equations over strings into account. Then the existing notion of redundancy and model construction techniques for the equational theorem proving framework for clauses over strings has been adapted. This paper has also provided a decision procedure for word problems over strings w.r.t. a set of conditional equations $R$ over strings if $R$ can be finitely saturated under the Superposition, Rewrite and Equality Resolution rule. (The complexity analysis of the proposed approach is not discussed in this paper. It is left as a future work for this decision procedure.)

Since strings are fundamental objects in mathematics, logic, and computer science including formal language theory, developing applications based on the proposed superposition calculus with strings may be a promising future research direction. Also, the results in this paper may have potential applications in verification systems and solving satisfiability problems [1].

In addition, it would be an interesting future research direction to extend our superposition calculus with strings to superposition calculi with strings using built-in equational theories, such as commutativity, *idempotency* [8], *nilpotency* [15], and their various combinations. For example, research on superposition theorem proving for *commutative monoids* [25] is one such direction.

[6] The reader is also encouraged to see *AVATAR modulo theories* [24], which is based on the concept of splitting.

## References


[1] Alessandro Armando, Silvio Ranise & Michael Rusinowitch (2003): *A rewriting approach to satisfiability procedures*. Information and Computation 183(2), pp. 140 – 164.

[2] Franz Baader & Tobias Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK.

[3] Leo Bachmair & Harald Ganzinger (1994): *Rewrite-based Equational Theorem Proving with Selection and Simplification*. J. Log. Comput. 4(3), pp. 217–247.

[4] Leo Bachmair & Harald Ganzinger (1995): *Associative-commutative superposition*. In Nachum Dershowitz & Naomi Lindenstrauss, editors: Conditional and Typed Rewriting Systems, Springer, Berlin, Heidelberg, pp. 1–14.

[5] Leo Bachmair & Harald Ganzinger (1998): *Equational Reasoning in Saturation-Based Theorem Proving*. In Wolfgang Bibel & Peter H. Schmitt, editors: Automated Deduction. A basis for applications, chapter 11, Volume I, Kluwer, Dordrecht, Netherlands, p. 353–397.

[6] Clark Barrett, Roberto Sebastiani, Sanjit A Seshia & Cesare Tinelli (2009): *Satisfiability Modulo Theories*. In: Handbook of satisfiability, Frontiers in Artificial Intelligence and Applications 185, IOS Press, pp. 825–885.

[7] Ronald V. Book & Colm P. O'Dunlaing (1981): *Testing for the Church-Rosser property*. Theoretical Computer Science 16(2), pp. 223–229.

[8] Ronald V. Book & Friedrich Otto (1993): *String-Rewriting Systems*. Springer, New York, NY.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein (2001): *Introduction to Algorithms*, second edition. The MIT Press.

[10] Robert Cremanns & Friedrich Otto (2002): *A Completion Procedure for Finitely Presented Groups That Is Based on Word Cycles*. J. Autom. Reason. 28(3), pp. 235–256.

[11] Thomas Deiß (1992): *Conditional Semi-Thue systems for Presenting Monoids*. In: Annual Symposium on Theoretical Aspects of Computer Science, STACS 1992, Springer, Berlin, Heidelberg, pp. 557–565.

[12] Nachum Dershowitz (1991): *Canonical sets of Horn clauses*. In Javier Leach Albert, Burkhard Monien & Mario Rodríguez Artalejo, editors: Automata, Languages and Programming, Springer Berlin Heidelberg, pp. 267–278.

[13] Nachum Dershowitz & David A. Plaisted (2001): *Rewriting*. In: Handbook of Automated Reasoning, chapter 9, Volume I, Elsevier, Amsterdam, pp. 535 – 610.

[14] David B. A. Epstein, Mike Paterson, James W. Cannon, Derek F. Holt, Silvio V. F. Levy & William Thurston (1992): *Word Processing in Groups*. A. K. Peters, Ltd., Natick, MA.

[15] Qing Guo, Paliath Narendran & David A. Wolfram (1996): *Unification and matching modulo nilpotence*. In Michael A. McRobbie & John K. Slaney, editors: The 13th International Conference on Automated Deduction (CADE-13), Lecture Notes in Computer Science 1104, Springer, Berlin, Heidelberg, pp. 261–274.

[16] Derek F. Holt, Bettina Eick & Eamonn A. O'Brien (2005): *Handbook of computational group theory*. CRC Press, Boca Raton, FL.

[17] Deepak Kapur & Paliath Narendran (1985): *The Knuth-Bendix Completion Procedure and Thue Systems*. SIAM Journal on Computing 14(4), pp. 1052–1072.

[18] Dohan Kim & Christopher Lynch (2021): *Equational Theorem Proving Modulo*. In: The 28th International Conference on Automated Deduction (CADE-28), Lecture Notes in Computer Science 12699, Springer, pp. 166–182.

[19] Temur Kutsia (2002): *Theorem Proving with Sequence Variables and Flexible Arity Symbols*. In Matthias Baaz & Andrei Voronkov, editors: Logic for Programming, Artificial Intelligence, and Reasoning, 9th International Conference, LPAR 2002, Tbilisi, Georgia, October 14-18, 2002, Proceedings, Lecture Notes in Computer Science 2514, Springer, pp. 278–291.

[20] Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett & Morgan Deters (2016): *An Efficient SMT Solver for String Constraints*. *Form. Methods Syst. Des.* 48(3), p. 206–234.

[21] Klaus Madlener, Paliath Narendran & Friedrich Otto (1991): *A Specialized Completion Procedure for Monadic String-Rewriting Systems Presenting Groups*. In Javier Leach Albert, Burkhard Monien & Mario Rodríguez-Artalejo, editors: *Automata, Languages and Programming, 18th International Colloquium, ICALP91, Madrid, Spain, July 8-12, 1991, Proceedings*, *Lecture Notes in Computer Science* 510, Springer, pp. 279–290.

[22] Robert Nieuwenhuis & Albert Rubio (2001): *Paramodulation-based theorem proving*. In: *Handbook of Automated Reasoning*, chapter 7, Volume I, Elsevier, Amsterdam, pp. 371–443.

[23] Friedrich Otto, Masashi Katsura & Yuji Kobayashi (1997): *Cross-Sections for Finitely Presented Monoids with Decidable Word Problems*. In Hubert Comon, editor: *Rewriting Techniques and Applications, 8th International Conference, RTA-97, Sitges, Spain, June 2-5, 1997, Proceedings*, *Lecture Notes in Computer Science* 1232, Springer, pp. 53–67.

[24] Giles Reger, Nikolaj Bjøner, Martin Suda & Andrei Voronkov (2016): *AVATAR Modulo Theories*. In Christoph Benzmüller, Geoff Sutcliffe & Raul Rojas, editors: *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, *EPiC Series in Computing* 41, pp. 39–52.

[25] José Carlos Rosales, Pedro A. García-Sánchez & Juan M. Urbano-Blanco (1999): *On Presentations of Commutative Monoids*. *International Journal of Algebra and Computation* 09(05), pp. 539–553.

[26] Albert Rubio (1996): *Theorem Proving modulo Associativity*. In Hans Kleine Büning, editor: *Computer Science Logic*, Springer, Berlin, Heidelberg, pp. 452–467.

# Nominal Sets in Agda
# A Fresh and Immature Mechanization

Miguel Pagano*
FAMAF - Universidad Nacional de Córdoba
Córdoba, Argentina
miguel.pagano@unc.edu.ar

José E. Solsona
Facultad de Ingeniería - Universidad ORT Uruguay
Montevideo, Uruguay
solsona@ort.edu.uy

In this paper we present our current development on a new formalization of nominal sets in Agda. Our first motivation in having another formalization was to understand better nominal sets and to have a playground for testing type systems based on nominal logic. Not surprisingly, we have independently built up the same hierarchy of types leading to nominal sets. We diverge from other formalizations in how to conceive finite permutations: in our formalization a finite permutation is a permutation (i.e. a bijection) whose domain is finite. Finite permutations have different representations, for instance as compositions of transpositions (the predominant in other formalizations) or compositions of disjoint cycles. We prove that these representations are equivalent and use them to normalize (up to composition order of independent transpositions) compositions of transpositions.

## 1 Introduction

Nominal sets were introduced to Computer Science by Gabbay and Pitts to give an adequate mathematical universe that permits the definition of inductive sets with binding [8]. Instead of taking equivalence classes of inductively defined sets (as in a formal treatment of, say, the Lambda Calculus) or a particular representation of the variables (as in the de Bruijn approach to Lambda Calculus), nominal sets have a notion of name abstraction that ensures all the properties expected for binders; in particular, alpha-equivalent lambda terms are represented by the same element of the nominal set of lambda terms.

In this paper we present a new mechanization [9] of nominal sets. Most of the current mechanizations of nominal sets represent finite permutations as compositions of transpositions, where transpositions are represented by pairs of atoms and compositions as lists. In contrast, our starting point is permutations (i.e. bijective functions); finite permutations are permutations that can be represented by composition of transpositions. Moreover they conflate the set of atoms mentioned in a list with twhe domain of the (represented) permutation. Pondering about this issue, we decided to develop a "normalization" procedure for representations of finite permutations; in order to prove its correctness, we were driven to introduce cycle notation.

The rest of this paper is structured into three sections. In Sect. 2 we summarize the fundamentals of Nominal Sets; then, in Sect. 3 we present the most salient aspects of our mechanization in Agda; and finally in Sect. 4 we conclude by mentioning related works and contrasting them with our approach, indicating also our next steps. We assume some knowledge of Agda, but also hope that the paper can be followed by someone familiar with any other language based on type theory.

*Most of this work was done in a research leave in ORT Uruguay, financed by Agencia Nacional de Investigación e Innovación (ANII) of Uruguay.

## 2 Fundamentals of Nominal Sets

In this section we summarize the main concepts underlying the notion of Nominal Sets; for a more complete treatment we refer the reader to [11]. We repeat the basic definitions of group and group action. A *group* is a set $G$ with a distinguished element ($\varepsilon \in G$, the *unit*), a binary operation ($\_\cdot\_ \colon G \times G \to G$, the *multiplication*), and a unary operation ($\_^{-1} \colon G \to G$, the *inverse*), satisfying the following axioms:

$$\varepsilon \cdot g = g = g \cdot \varepsilon \qquad g \cdot (g^{-1}) = \varepsilon = g^{-1} \cdot g \qquad g_1 \cdot (g_2 \cdot g_3) = (g_1 \cdot g_2) \cdot g_3$$

Although a group is given by the tuple $(G, \varepsilon, \_\cdot\_, \_^{-1})$ (and the proofs that these operations satisfy the axioms) we will refer to the group simply by $G$. A sub-group of $G$ is a subset $H \subseteq G$ such that $\varepsilon \in H$ and $H$ is closed under the inverse and multiplication.

Let $G$ be a group. A *$G$-set* is a set $X$ with an operation $\_\bullet\_ \colon G \times X \to X$ (called the *action*) satisfying:

$$\varepsilon \bullet x = x \qquad g \bullet (g' \bullet x) = (g \cdot g') \bullet x$$

A morphism between $G$-sets $X$ and $Y$ is a function $F \colon X \to Y$ that commutes with the actions:

$$F\,(g \bullet x) = g \bullet F\,x$$

These are called *equivariant* functions. Since $id_X$ is equivariant and the composition of equivariant functions yields an equivariant function we can talk of the category of $G$-Sets.

Any set $X$ can be seen as a $G$-set by letting $g \bullet x = x$; such a $G$-set is called the *discrete* $G$-set. Moreover any group acts on itself by the multiplication.

One can form the (in)finitary product of $G$-sets by defining the action of $g$ on a tuple in a pointwise manner: $g \bullet \langle t, t' \rangle = \langle g \bullet t, g \bullet t' \rangle$; the projections and the product morphism $\langle F, H \rangle$ are equivariant, assuming that $F$ and $H$ are also equivariant. $G$-set, as a category, also has co-products.

If $X$ and $Y$ are $G$-sets one can endow the set $Y^X$ of functions from $X$ to $Y$ with the *conjugate* action: $(g \bullet F)\,x = g \bullet (F\,(g^{-1} \bullet x))$.

**$G$-sets over the Permutation Group** The group of symmetries over a set $\mathbb{A}$ consists of $G = Sym(\mathbb{A})$, where $Sym(\mathbb{A})$ is the set of bijections on $\mathbb{A}$; the multiplication of $Sym(\mathbb{A})$ is composition, the inverse is the inverse bijection, and the unit is the identity.

Let $Perm(\mathbb{A})$ be the subset of $Sym(\mathbb{A})$ of bijections that changes only finitely many elements; i.e., $f \in Perm(\mathbb{A})$ if $\mathrm{supp}(f) = \{a \in \mathbb{A} \mid f\,a \neq a\}$ is finite. It is straightforward to prove that $Perm(\mathbb{A})$ is a sub-group of $Sym(\mathbb{A})$. Of course, if $\mathbb{A}$ is finite, then $Perm(\mathbb{A}) = Sym(\mathbb{A})$. Notice that $\mathbb{A}$ is a $Perm(\mathbb{A})$-set with the action being function application: $\pi \bullet a = \pi\,a$. A basic result is that every finite permutation can be decomposed as the composition of *transpositions*: if $a, a' \in \mathbb{A}$, then $(a\ a')$ is the permutation that swaps $a$ with $a'$ and is the identity in $\mathbb{A} \setminus \{a, a'\}$.

**Nominal Sets** Let $X$ be a $Perm(\mathbb{A})$-set, we say that $x \colon X$ *is supported by* $A \subseteq \mathbb{A}$ if

$$\forall \pi.(\forall a \in A.\pi\,a = a) \implies \pi \bullet x = x \ .$$

We say that $X$ is a *nominal set* if each element of $X$ is supported by some finite subset of $\mathbb{A}$. Since each finite permutation can be decomposed as a composition of transpositions, then one can prove that the above definition is equivalent to

$$\forall a, a' \in \mathbb{A} \setminus A.(a\ a') \bullet x = x \ .$$

In his book [11] Pitts uses classical logic to prove that if $x$ is supported by some finite set $A$, then there exists a least supporting set, called *the* support of $x$. As shown by Swan [12] one cannot define the least support in a constructive setting; therefore a formalization in a constructive type theory should ask for "some" finite support. This affects the notion of freshness: in classical logic, $x$ *is fresh for* $y$ if $\mathrm{supp}(x) \cap \mathrm{supp}(y) = \emptyset$, with $x \in X$ and $y \in Y$ being elements of different nominal sets; in a constructive setting one has to limit this relation to atoms: $a \in \mathbb{A}$ is fresh for $x \in X$ if $a \notin \mathrm{supp}(x)$, where $\mathrm{supp}(x)$ is the set supporting $x$, not necessarily the least one. Notice that the definition is the same ("there exists some finite support for each element"), but in classical logic that is sufficient to obtain the least support.

## 3 Our Formalization in Agda

Our formalization is developed on top of the Agda's standard library [13]. The standard library includes an algebraic structure going beyond groups; it lacks, however, a formalization of group actions. We present first the definition of Group in the standard library:

```
record Group c ℓ : Set (suc (c ⊔ ℓ)) where
  field
    Carrier : Set c
    _≈_     : Rel Carrier ℓ
    _·_     : Op₂ Carrier
    ε       : Carrier
    _⁻¹     : Op₁ Carrier
    isGroup : IsGroup _≈_ _·_ ε _⁻¹
```

A Group is a *bundle* where the components of its definition (the carrier set, the unit, the inverse, the composition) are explicitly mentioned plus a proof *isGroup* that they satisfy the axioms. Notice that one of the fields is a relation `_≈_`; that relation should be an equivalence relation over the carrier: essentially this amounts to say that the `Carrier` has a setoid structure. Setoids allows for greater flexibility as they enable to work with a notion of equality that is not the propositional equality; `Func A B` is the set of functions between setoids `A` and `B` that preserve the equality.

**G-Sets** Our first definition is the *structure* that collects the equations required for an action. In the following, we are under a module parameterized by `G : Group`.

```
record IsAction (F : Func (G.setoid ×ₛ A) A) : Set _ where
  _·ₐ_ : Carrier G → Carrier A → Carrier A
  g ·ₐ x = Func.f F (g , x)
  field
    idₐ   : ∀ x → ε ·ₐ x ≈A x
    compₐ : ∀ g' g x → g' ·ₐ g ·ₐ x ≈A (g' · g) ·ₐ x
```

Notice that the record-type `IsAction` is a predicate over functions from the setoid $G \times A$ to $A$. The definition of GSet is straightforward and follows the pattern of the standard library:

```
record GSet : Set _ where
  field
    set : Setoid ℓ₁ ℓ₂
    action : Func (G.setoid ×ₛ set) set
    isAction : IsAction action
```

The next concept is that of equivariant function.[1]

```
record Equivariant  (A : GSet) (B : GSet) : Set _ where
  field
    F : Func (set A) (set B)
    isEquivariant : IsEquivariant (action A) (action B) F
```

**Permutations** A finite permutation can be given by a bijective map, as a composition of transpositions, or as a composition of disjoint cycles. Let us exhibit this with a concrete example; let $f\colon \mathbb{N} \to \mathbb{N}$ be defined as $fx = (x+2)\,mod\,6$ if $x < 6$ and $fx = x$ if $x \geqslant 6$; it can be expressed as the composition of two cycles: $(1\,3\,5)\ (0\,2\,4)$. Alternatively, it can also be expressed as composition of transpositions $(1\,3)\,(3\,5)\,(0\,2)\,(2\,4)$.

Transpositions are defined over a decidable setoid; in the following we assume that `A-setoid` has type `DecSetoid` $\ell\ \ell'$, and $\_\stackrel{?}{=}\_$ decides the equality.

```
  A = Carrier A-setoid ; _≈A_ = _≈_ A ; Perm = Inverse A-setoid A-setoid
  transp : A → A → A → A
  transp a b c with does (c ≟ a)
  ... | true = b
  ... | false with does (c ≟ b)
  ... | true = a
  ... | false = c
```

We started with the following syntactic representation of Finite Permutations:

```
  _≈ₚ_ : Rel Perm _
  F ≈ₚ G = (x : Carrier A) → f F x ≈A f G x
data FinPerm : Set ℓ where
  Id : FinPerm
  Comp : (p q : FinPerm) → FinPerm
  Swap : (a b : A) → FinPerm
```

In order to define the bijection represented by `p : FinPerm` we use `transp` and its properties; here we need to ask the `A-setoid` to be decidable.

```
⟦_⟧ : FinPerm → Perm
⟦ Id ⟧ = idₚ setoid
⟦ Comp p q ⟧ = ⟦ q ⟧ ∘ₚ ⟦ p ⟧ -- _∘ₚ_ is the composition of Perm
⟦ Swap a b ⟧ = record {
    f = transp a b  ; f⁻¹ = transp a b
  ; cong₁ = transp-respects-≈ a b ; cong₂ = transp-respects-≈ a b
  ; inverse = transp-involutive a b , transp-involutive a b }
```

The decidability of the equivalence of `A-setoid` implies the decidability of finite permutations:

```
_≟ₚ_ : ∀ p q → Dec (⟦ p ⟧ ≈ₚ ⟦ q ⟧)
```

The point is that one does not need to check all the atoms but only those in the support of `p`. Moreover we can define a correct "normalization" procedure to get a permutation equal to `p` but without any redundant transposition (and at most one `Id`), passing through a representation by cycles:

[1] We note that Choudhury defines equivariant functions only for the group of finite permutations; it seems absent in the mechanization of Paranhos and Ventura.

```
  norm : FinPerm → FinPerm
  norm = cycles-to-FP ∘ cycles-from-FP
  norm-corr : ∀ p → ⟦ p ⟧ ≈ₚ ⟦ norm p ⟧
```

The functions `cycles-to-FP` maps lists of disjoint cycles to `FinPerm` and `cycles-from-FP` goes in the reverse direction, producing a list of disjoint cycles from a `FinPerm`.

Let us remark that `FinPerm` is just a representation and the set of finite permutation, `PERM`, is the subset of `Perm` corresponding to the image of $[\![\_]\!]$:

```
PERM : Set _
PERM = Σ[ p ∈ Perm ] (Σ[ q ∈ FinPerm ] (p ≈ₚ ⟦ q ⟧))
```

**Nominal Sets** Remember that a subset $A \subseteq \mathbb{A}$ is a support for $x$ if every permutation fixing every element of $A$ fixes $x$, through the action. A subset of a setoid `A` can be defined either as a predicate or as pairs (just as in `PERM` where the predicate is `λ p → Σ[ q ∈ FinPerm ] (p ≈ₚ ⟦ q ⟧)`) or as another type, say `B`, together with an injection `ι : Injection B A`.

```
variable
  X : GSet
  P : SetoidPredicate A-setoid
is-supp : Pred X _
is-supp x = (π : PERM) → (predicate P ⊆ _∉-dom (proj₁ π)) → (π ·ₐ x) ≈X x
```

The predicate `λ a → f (proj₁ π) a ≈A a` is `_∉-dom (proj₁ π)`; therefore, if `P a` iff $a \in A$, then `predicate P ⊆ _∉-dom (proj₁ π)` is a correct formalization of $\forall a \in A.\pi\, a = a$.

Our official definition of support is the following:

```
  _supports_ : Pred X _
  _supports_ x = ∀ {a b} → a ∉ₛ P → b ∉ₛ P → SWAP a b ·ₐ x ≈X x
```

Here `SWAP` is a `PERM`utation equal to $[\![$`Swap a b`$]\!]$. We formally proved that both definitions are equivalent.

In order to define nominal sets we need to choose how to say that a subset is finite; as explained by Coquand and Spiwak [7] there are several possibilities for this. We choose the easiest one: a predicate is finite if there is a list that enumerates all the elements satisfying the predicate.

```
finite : Pred (SetoidPredicate setoid) _
finite P = Σ[ as ∈ List Carrier ] (predicate P ⊆ (_∈ as))
```

A GSet is nominal if all the elements of the underlying set are finitely supported.

```
record Nominal (X : GSet) : Set _ where
  field
    sup : ∀ x → Σ[ P ∈ SetoidPredicate setoid ] (finite P × P supports x)
```

It is easy to prove that various constructions are nominals; for instance any discrete GSet is nominal because every element is supported by the empty predicate $\bot_s$:

```
Δ-nominal : (S : Setoid _ _) → Nominal (Δ S)
sup (Δ-nominal S) x = ⊥ₛ , ⊥-finite , (λ _ _ → S-refl {x = x})
  where open Setoid S renaming (refl to S-refl)
```

We have defined `GSet-⇒ A B` corresponding to the GSet of equivariant functions from `A` to `B`; now we can prove that `GSet-⇒ A B` is nominal, again with $\bot_s$ as the support for any `F : Equivariant A B`.

```
  →-nominal : Nominal (GSet-⇒ A B)
  sup (→-nominal) F = ⊥ₛ , ⊥-finite , λ _ _ → supported
    where
    supported : ∀ {a b} x → f ((SWAP a b) ·→ F) x ≈B f F x
```

# 4 Conclusion

Nominal techniques have been adopted in various developments. We distinguish developments borrowing some concepts from nominal techniques to be applied in specific use cases (e.g. formalization of languages with binders like the $\lambda$ or $\pi$ calculus with their associated meta-theory) [2, 6, 5, 4] from more general developments aiming to formalize at least the core aspects of the theory of nominal sets. We are more concerned with the later type.

The nominal datatype package for Isabelle/HOL [14] developed by Urban and Berghofer implements an infrastructure for defining languages involving binders and for reasoning conveniently about alpha-equivalence classes. This Isabelle/HOL package inspired Aydemir et al. [1] to develop a proof of concept for the Coq proof assistant, however it had no further development. In his Master thesis [3], Choudhury notes that none of the previous developments following the theory of nominal sets were based on constructive foundations. He showed that a considerable portion (most of the first four chapters of Pitts book [11]) of the theory of nominal sets can also be developed constructively by giving a formalization in Agda. Pitts original work is based on classical logic, and depends heavily on the existence of the smallest finite support for an element of a nominal set. However, Swan [12] has shown that in general this existence cannot be constructively guaranteed, as it would imply the law of the excluded middle.

Choudhury works with the notion of *some non-unique support*. In order to formalize the category of Nominal Sets, Choudhury preferred setoids instead of postulating functional extensionality. As far as we know, Choudhury is still the most comprehensive mechanization in terms of instances of constructions having a nominal structure.

Recently Paranhos and Ventura [10] presented a constructive formalization in Coq of the core notions of nominal sets: support, freshness and name abstraction. They follow closely Choudhury's work in Agda [3], acknowledging the importance of working with setoids. They claim that by using Coq's type class and setoid rewriting mechanism, much shorter and simpler proofs are achieved, circumventing the "setoid hell" described by Choudhury.

Both of those two formalizations in type theory take a very pragmatic approach to finite permutations: a finite permutation is a list of pairs of names. In our approach, we start with the more general notion of bijective function from which the finite permutations are obtained as a special case; moreover having different representations allowed us to state and prove some theorems that cannot even be stated in the other formalizations. So far, our main contributions are: the representation of finite permutations and the normalization of composition of transpositions; the equivalence between two definitions of the relation "$A$ supports the element $x$"; and proving that the extension of every container type can be enriched with a group action (notice that this cover lists, trees, etc.).

Our next steps are the definition of freshness, we are studying an alternative notion of support that would admit having a freshness relation between elements of two nominal sets (in contrast with other mechanization that only consider "the atom $a$ is fresh for $x$") and name abstraction. In parallel we hope to be able to prove that extensions of finite containers on nominal sets are also nominal sets. We also hope to streamline further some rough corners of our development.

## Acknowledgments

This formalization grew up from discussions with the group of the research project "Type-checking for a Nominal Type Theory": Maribel Fernández, Nora Szasz, Álvaro Tasistro, and Sebastián Urciouli. We thank Cristian Vay for discussions about group theory. This work was partially funded by Agencia Nacional de Investigación e Innovación (ANII) of Uruguay.

# References


[1] Brian E. Aydemir, Aaron Bohannon & Stephanie Weirich (2007): *Nominal Reasoning Techniques in Coq: (Extended Abstract)*. *Electron. Notes Theor. Comput. Sci.* 174(5), pp. 69–77, doi:10.1016/j.entcs.2007.01.028.

[2] Jesper Bengtson & Joachim Parrow (2009): *Formalising the pi-calculus using nominal logic*. *Log. Methods Comput. Sci.* 5(2). Available at `http://arxiv.org/abs/0809.3960`.

[3] Pritam Choudhury (2015): *Constructive Representation of Nominal Sets in Agda*. Master's thesis, Cambridge University.

[4] Ernesto Copello, Nora Szasz & Álvaro Tasistro (2018): *Formalisation in Constructive Type Theory of Barendregt's Variable Convention for Generic Structures with Binders*. *Electronic Proceedings in Theoretical Computer Science* 274.

[5] Ernesto Copello, Nora Szasz & Álvaro Tasistro (2018): *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, pp. 79–95.

[6] Ernesto Copello, Álvaro Tasistro, Nora Szasz, Ana Bove & Maribel Fernández (2016): *Alpha-Structural Induction and Recursion for the Lambda Calculus in Constructive Type Theory*. *Electronic Notes in Theoretical Computer Science* 323, pp. 109–124. Proceedings of the Tenth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2015).

[7] Thierry Coquand & Arnaud Spiwack (2010): *Constructively finite?* In: *Contribuciones científicas en honor de Mirian Andrés Gómez*, Universidad de La Rioja, pp. 217–230.

[8] M. J. Gabbay & A. M. Pitts (2002): *A New Approach to Abstract Syntax with Variable Binding*. *Formal Aspects of Computing* 13, pp. 341–363.

[9] Miguel Pagano & José E. Solsona (2022): *Nominal Sets in Agda*. `https://github.com/miguelpagano/nominal-sets/`.

[10] Fabrício S. Paranhos & Daniel Ventura: *Towards a Formalization of Nominal Sets in Coq*. `https://popl22.sigplan.org/details/CoqPL-2022-papers/4/Towards-a-Formalization-of-Nominal-Sets-in-Coq`. Online; accessed 1 May 2022.

[11] Andrew M. Pitts (2013): *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, England.

[12] Andrew Swan (2017): *Some Brouwerian Counterexamples Regarding Nominal Sets in Constructive Set Theory*, doi:10.48550/ARXIV.1702.01556. Available at `https://arxiv.org/abs/1702.01556`.

[13] The Agda Team (2021): *The Agda standard library, version 1.7*. `https://github.com/agda/agda-stdlib`.

[14] Christian Urban & Stefan Berghofer (2006): *A Recursion Combinator for Nominal Datatypes Implemented in Isabelle/HOL*. In Ulrich Furbach & Natarajan Shankar, editors: *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, *Lecture Notes in Computer Science* 4130, Springer, pp. 498–512, doi:10.1007/11814771_41. Available at `https://doi.org/10.1007/11814771_41`.

# Tool support for interval specifications in differential dynamic logic*

Jaime Santos
Universidade do Minho, Braga, Portugal
CIDMA, Univ. Aveiro, Portugal

Alexandre Madeira
CIDMA, Univ. Aveiro, Portugal

Daniel Figueiredo
CIDMA, Univ. Aveiro, Portugal

A wide range of methods and techniques from computer science, and particularly from systems modeling and verification, is being applied to many modern engineering domains, including synthetic biology.

Most behaviors described in synthetic biology have a hybrid nature, in the sense that both discrete or continuous evolutions are observed. Differential Dynamic Logic ($d\mathscr{L}$) is a well-known formalism used for the rigorous treatment of these systems by considering variables whose valuations change through differential equations and discrete assignments. Since the models of synthetic biology are usually described by ranges of values, due to errors and perturbations of observed quantities, recent work within the team proposed an interval version of $d\mathscr{L}$, where variables are interpreted as intervals.

This paper presents the first steps in the development of computational support for this formalism. In this view, we introduce a tool to build models based on intervals, prepared to translate them into specifications ready to be processed by the KeYmaera X tool.

## 1 Introduction and preliminaries

Hybrid systems – those composed of continuous and discrete components – are, nowadays, everywhere, from the medical devices we use to the aerospace artifacts we have. Due to the critical role that some of them play in our life, the scientific community was pushed to develop theories and tools to support the trustworthy conception of these systems, not only via simulation techniques (e.g. with `Simulink`) but also using program verification techniques and logic.

Differential Dynamic Logic [10], with its supporting tool `KeYmaera X` [9], represents a core formalism in this context. This approach brings principles and techniques from program verification to hybrid systems developers, namely from dynamic logic [7]. The formalism has been successfully applied to several computational hybrid systems scenarios such as plane traffic, surgical robots and automotive cruise control [10]; but also in other less obvious domains, including the specification and analysis of models in biology [6].

Differential dynamic logic enriches dynamic logic by expanding the structure of programs with two kind of atomic programs: basic assignments $x := \theta$ for discrete state transitions; and continuous evolutions of states $x' := \theta \,\&\, \chi$. This syntax can be used to describe the dynamics of discrete systems, continuous systems and even combine both kinds of atomic programs to express and prove the correctness of assertions which admit both (the so-called cyber-physical or hybrid systems). Because of this, a programs of $d\mathscr{L}$ is called a *hybrid programs*. Then, with the sound proof calculus for $d\mathscr{L}$, and with the `KeYmaera X` tool – a semi-automatic prover – we can prove the correctness of such propositions. When the user provides a formula of $d\mathscr{L}$ as input, `KeYmaera X` either generates its proof or retrieves some simpler formulas - preconditions - that are required to be valid to prove the formula.

*This work is supported by by FCT, the Portuguese funding agency for Science and Technology by CIDMA (UIDB/04106/2020), and by the project IBEX (PTDC/CCI-COM/4280/2021), .

The syntax of $d\mathscr{L}$ can be seen as the one of a first-order logic embedding by a dynamic logic. We can consider a first-order fragment of $d\mathscr{L}$, where the set of formulas do not contain modalities and which is denoted by $Fml_{FOL}(\Sigma, X)$.

**Definition 1** *Let $\Sigma$ be a signature containing n-ary functions, propositions, and state variables; and $X$ be a set of logical variables.*

- *The set $Trm(X,\Sigma)$ of terms is the least set containing $X$ such that $f(t_1,...,t_n) \in Trm(X,\Sigma)$ iff $f \in \Sigma$ is a n-ary function and $t_1,...,t_n \in Trm(X,\Sigma)$;*
- *$p(t_1,...,t_n)$ is a predicate if $p \in \Sigma$ is a n-ary proposition and $t_1,...,t_n \in Trm(\Sigma,X)$;*
- *$Fml_{FOL}(X,\Sigma)$ is the least set containing every predicate, and such that $\varphi \vee \psi$, $\varphi \wedge \psi$, $\neg\varphi$, $\forall\varphi$, $\exists\varphi \in Fml_{FOL}(X,\Sigma)$ whenever $\varphi, \psi \in Fml_{FOL}(X,\Sigma)$.*

Note that constants are 0-ary functions in $\Sigma$ and that other Boolean operators – such as $\rightarrow$ – can be introduced as usual abbreviations. State variables are distinct from logical variables because the value of state variables may be affected by the programs within modalities, as explained further. We denote by $\Sigma_{fl}$ the set of state variables.

Here, it is important to mention that the semantics of $d\mathscr{L}$ are evaluated over the reals. Furthermore, a strict interpretation of formulas and propositions from $\Sigma$ is imposed. This mean that $\Sigma$ only contain functions such as $+$ or $power(\cdot,\cdot)$ that must be interpreted as "sum" or "exponentiation", respectively. Similarly, $\Sigma$ can only include propositions like $=$ or $\leq$ which are to be interpreted as "equal" and "less or equal", accordingly.

In addition to the usual Boolean operators and quantifiers, the whole set of formulas of $d\mathscr{L}$ makes use of modalities labeled with some hybrid program $\alpha$. These hybrid programs can be used to describe the dynamics of discrete, continuous, or even hybrid systems.

**Definition 2** *The whole set $HP(X,\Sigma)$ of hybrid programs is defined as follows:*

- *$(x_1 := t_1,...,x_n := t_n)$, $(x_1' = t_1,...,x_n' = t_n \,\&\, \psi)$, $?\psi \in HP(X,\Sigma)$ for every state variables $x_1,...,x_n \in \Sigma$, $t_1,...,t_n \in Trm(X,\Sigma)$ and $\psi \in Fml_{FOL}(X,\Sigma)$;*
- *$\alpha;\beta$, $\alpha \cup \beta$, $\alpha^* \in HP(X,\Sigma)$ whenever $\alpha, \beta \in HP(X,\Sigma)$.*

**Definition 3** *The full set $Fml(X,\Sigma)$ of formulas of $d\mathscr{L}$ is defined recursively as the least set containing $Fml_{FOL}(X,\Sigma)$ and such that:*

- *$[\alpha]\varphi$, $\langle\alpha\rangle\varphi \in Fml(X,\Sigma)$ whenever $\varphi \in Fml(X,\Sigma)$ and $\alpha \in HP(X,\Sigma)$*
- *$\varphi \vee \psi$, $\varphi \wedge \psi$, $\neg\varphi$, $\forall\varphi$, $\exists\varphi \in Fml(X,\Sigma)$ whenever $\varphi, \psi \in Fml(X,\Sigma)$.*

The truth of a formula in the scope of a modality embedding a hybrid program is evaluated in a scenario obtained after the hybrid program is run. Note the difference between the two modalities: $[\alpha]\varphi$ is *true* if $\varphi$ holds at every scenario reached by the hybrid program $\alpha$; and $\langle\alpha\rangle\varphi$ is *true* if $\varphi$ holds at some scenario reached by the hybrid program $\alpha$. Thus, we can encode properties such as *"from a point where $x = 0$ it is always true that running a program that makes $x$ grow with a rate of two units per unit of time, $x$ will be positive"* with the formula $x = 0 \rightarrow [x' = 2]x \geq 0$. Similarly, we can translate the property *"whatever the initial value of $y$, we will eventually obtain a positive value if we repeatedly add 1 to $y$"* into the formula $\langle(y := y+1)^*\rangle\, y > 0$.

**Example 1 ((Bouncing ball from [10]))** *Let us consider a ball that is dropped from some height H and bounces back when it hits the ground. Denoting by g the gravity acceleration constant (function with arity 0), we can describe the movement of the ball in the air by the following atomic program:* $(h' = v, v' = h \,\&\, h \geq 0)$. *Note that,* $h, v$ *are state variables, and the constraint* $h \geq 0$ *(the ball will not go below the ground)*

*To fully represent the dynamics of this system, we also need to consider the case when the ball hits the ground. In this case, letting* $c \in ]0,1[$ *be the elasticity constant of the ball, we describe the discrete change in the velocity of the ball by the discrete assignment:* $v := -cv$.

*Thus, we can combine both atomic programs and describe the dynamics of the bouncing ball by a hybrid program* $\alpha$, *abbreviating* $((h' = v, v' = h \,\&\, h \geq 0) \cup (?h = 0; v := -cv))^*$. *Note that the operator* $^*$ *is added to allow the ball to bounce several times on the ground.*

*Finally, we can now express a property such as "the ball will never bounce higher than the initial height" by the formula* $[\alpha]\, h \leq H$.

Regarding $d\mathscr{L}$, three functions are used to interpret formulas: an interpretation $I$ – for rigid symbols in $\Sigma \backslash \Sigma_{fl}$; an assignment $\eta$ – for logical variables in $X$; and a state $v$ – for state variables in $\Sigma_{fl}$. A formula is said to be satisfiable if it is *true* for some triple $(I, \eta, v)$ and it is valid if it is *true* for every triple $(I, \eta, v)$. The full semantics can be found in [10] and will not be fully presented here as they can be derived as a particular case from the interval semantics. In [4], an interval syntax is developed for $d\mathscr{L}$, regarding its application in contexts where variables are presented in terms of intervals, namely due to errors or uncertainty. This can be useful, for instance, whenever we want to model a physical system and there is some uncertainty in a position $pos = [a,b]$ or whenever we want to make an assignment of an irrational number without machine representation. Apart from replacing real numbers for closed intervals, the interval syntax is the same as in $d\mathscr{L}$, and the semantics presented are adapted to an interval context, namely following the work of Moore [8]. As explained in [4], note that this interval syntax is not intended to introduce a formal translation but rather to introduce some abbreviations to $d\mathscr{L}$ syntax. For instance, real numbers are considered as degenerated intervals of the form $[a,a]$ for $a \in \mathbb{R}$. Under this perspective, the semantics for this adapted syntax considers a "strict" interpretation over closed intervals, *i.e.* a symbol like $+ \in \Sigma$ is interpreted as "interval sum" , for instance (check [8] for additional information in interval arithmetics). Also, logical and state variables are evaluated over $\mathscr{I}(\mathbb{R})$. Thus, the interpretation of each predicate $P$ (defined for reals) must be adapted to intervals. Particularly, denoting by $P^{\mathscr{I}}(\mathbb{R})$ the interval interpretation of $P$, a predicate $P^{\mathscr{I}}(\mathbb{R})(X_1, ...X_n)$ is said to be *true* if $P(x_1, ..., x_n)$ holds for every $(x_1, ..., x_n) \in X_1 \times ... \times X_n$. With this definition, we keep the coherence in such a way that the semantics of $d\mathscr{L}$ can be seen as a particular case of the interval one since the interpretation of its formulas is done over real numbers (the set of degenerated intervals).

The semantics of $d\mathscr{L}$ to evaluate the truth of formulas is straightforwardly adapted to its interval version, namely regarding Boolean operators, quantifiers, and modalities. Nevertheless, the main difference is observed in hybrid programs, namely at continuous evolutions (differential equations constrained by a first-order formula $\chi$). Given an initial state $u$, a system of differential equations $\overrightarrow{x}' = (f_1(\overrightarrow{x}), ..., f_n(\overrightarrow{x}))$ and a first-order formula $\chi$, the set of reachable states is obtained by computing the solution $F(t) = (F_1(t), ..., F_n(t))$ of the differential equation $f$ whose initial conditions are set by the state $u$. For each $\bar{t} \in \mathbb{R}_0^+$ we can define a reachable state $v$ according to $F^{\mathscr{I}(\mathbb{R})}(\bar{t})$ and $\chi$ in such a way that $b \in \mathbb{R}^n \in F^{\mathscr{I}(\mathbb{R})}(\bar{t})$ if there is an initial state $a \in \mathbb{R}^n$ such that $F(\bar{t}) = b$ and $F(t)$ satisfies $\chi$ for every $t \in [0, \bar{t}]$. This definition verifies two important properties known as *correctness* and *optimality* because of the continuity of $F$ (see [4] for more details).

This paper presents a parser for this interval syntax and a translator which accepts interval $d\mathscr{L}$

formulas and retrieves equivalent ones in standard $d\mathscr{L}$. In this way, we take advantage of the semantical soundness of $d\mathscr{L}$ and we can call on $d\mathscr{L}$ proof calculus. In particular, we can use `KeYmaera X` and try to obtain proof for the original interval $d\mathscr{L}$ formula. We illustrate the framework by modeling a biological regulatory network [5] – where there is a great level of uncertainty and variables like concentration of a protein or other components are rather expressed in intervals than with a determined value. For this example, we consider a formula in the interval syntax of $d\mathscr{L}$, describing a property of the system, and use our parser and translator to obtain its equivalent formula in $d\mathscr{L}$ standard syntax. We then use the automatic tactic of `KeYmaera X` to prove the correctness of this example.

## 2 The `idDL2dDL` tool

This section introduces a tool to parse and translate specifications from interval $d\mathscr{L}$ to specifications in standard $d\mathscr{L}$, following the theoretical work in [4]. The implementation, developed in *Python*, is structured in five main parts – the *lexer*, the *parser*, the *translator*, a *graphical user interface* (GUI), and the *interpreter*. Detailed user instructions are available in the GitHub repository[1].

The lexer, as the name suggests, performs the lexical analysis of the expressions provided as input, which is the process of converting a sequence of characters into *tokens*. This component is implemented by comparing the current character to a predefined list of allowed symbols and either generating and appending the token to a list, or returning an error. It is also crucial that the order of the inputted text is preserved, therefore our lexer has an attribute in the form of a *Position* class, which advances its position every time a new token is successfully created.

The parser is then called to perform a syntactic analysis over these tokens, returning an abstract syntax tree (AST) as output. The AST is composed of nodes, and the different types of nodes can be at different depths of the tree, according to the priority degrees of the operations. The nodes were constructed following the Definition 2 of hybrid programs, based on the work of [10]. The degrees of priority are then obtained by a series of binary operations between tokens, which ultimately reduce to atomic types that produce the proper nodes in the right order, or return an error if an illegal operation is detected. This was one of the core challenges with the parser, since it required some language design and testing to assure nodes are only created via legal operations.

The translator allows for conversion of interval dynamic logic expressions into regular $d\mathscr{L}$ formulas. This is composed of several *visit* methods for each type of node. These methods, when called, evaluate the tokens inside the node and convert them accordingly, preserving the priority degree of the AST. For example, when a token of the interval type is detected inside a node, an object of the *TranslatedInterval* class is tasked with creating an inequation between a fresh variable and the lower and upper bounds of the interval. However, since we want to produce a syntax that `KeYmaera X` can compute, many other expressions are converted according to `KeYmaera X`'s specifications. One of the main challenges with this component was ensuring that each detected interval generated an unique variable and that the final expression mapped the variable to the correct interval. For this purpose, we created a method capable of generating an infinite sequence of unique strings depending on a class attribute that is incremented every time an interval is visited. However, our implementation has a surprising performance cost, which will need to be addressed in the future.

The graphical user interface was implemented with Python's *tkinter* library, and is composed of a dedicated translation page and a translation history page. The first formula of Fig. 1 illustrates the direct

[1] github.com/JaimePSantos/idDL2DL

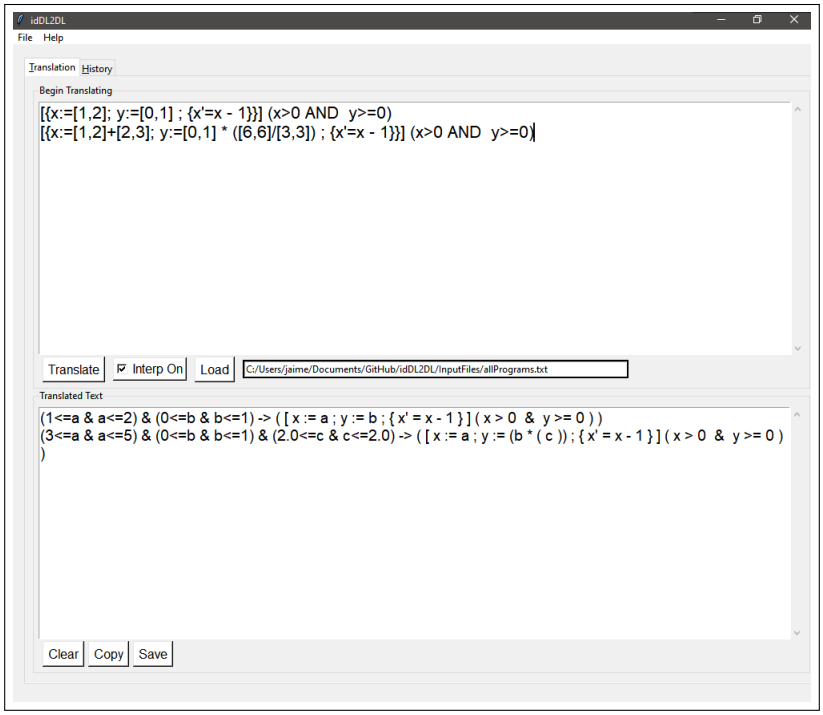



Figure 1: Translation of a simple example in `idDL2dDL`.

translation of the interval dynamic logic statement

```
[{x:=[1,2]; y:=[0,1] ; {x'=x-1}}] (x>0 AND y>=0)
```

to the respective differential dynamic logic one, ready to be analyzed in `KeYmaera X` - whose correctness is further proved using the default automatic proof tactic of `KeYmaera X`. The translation page also allows the user to load a file containing an arbitrary number of formulae and save the translated result directly to $*.kyx$ files. If only a single formula is translated, then the GUI prompts the user to enter a file name and location. If, however, multiple lines of translation are detected, the user will have the option to save all the translations in a single $*.kyx$ file, or generate multiple files for each translated line. The interface also supports a history page, where all the statements translated during the work session are saved. This feature is useful in aiding the stepwise analysis of more complex models. Displaying translation errors in a user-friendly way is a bit more demanding with this component, which is why this is still a work in progress.

## Interpreter and performance

Lastly, an initial version of an interpreter is the latest feature added to the software. This component, when enabled, receives an AST as input, and returns an interpreted string as output. The implementation logic is again to visit each node and depending on the type of operation associated with the node and

apply one of the following interval arithmetic rules of [8]

$$[a,b]+[c,d]=[a+c,b+d] \tag{1}$$
$$[a,b]-[c,d]=[a-d,b-c] \tag{2}$$
$$[a,b]\times[c,d]=[min(P),max(P)] \text{ where } P=\{a\times c,a\times d,b\times c,b\times d\} \tag{3}$$
$$[a,b]\div[c,d]=[min(P),max(P)] \text{ where } P=\{a\div c,a\div d,b\div c,b\div d\}, \quad 0\notin[c,d]. \tag{4}$$

The second formula of Fig. 1 presents an example of an interpreted and translated formula. Had the formula not gone through the interpretation process, it would require a fresh variable for each interval, meaning the translation would contain five variables instead of the three present in the figure. However, because we are visiting the nodes and interpreting them, the structure of the AST must be rebuilt with the resulting output. For this reason, the current version of the program needs to recreate and re-parse the tokens before feeding them to the translator, resulting in an inherent performance cost. Fig. 2 was

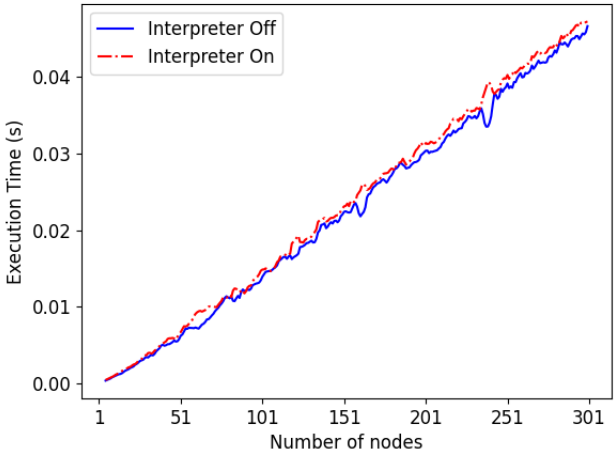



Figure 2: Comparison between the execution time with and without the interpreter, as a function of the number of nodes.

obtained by running the same formula, for a simple translation and an interpreted translation. The nodes were defined as the sum of consecutive divisions between two intervals, and each formula was sampled 100 times to plot the mean of each execution. This figure shows that the interpreter indeed adds a small performance cost, however it was expected to be significantly larger since the second round of lexing and parsing is required. This implies that the true bottleneck lies in the translation process, most likely due to an inefficient method of generating unique variables.

The implementation of this component had its own set of challenges, and the main one was ensuring that the visit methods correctly apply the defined operations, while ignoring and preserving nodes that contain operations between intervals and non-intervals. Currently, there is still a limitation on the interpreter since operations between intervals and expressions inside the parenthesis are kept separate, as can be seen in the translation of the second formula of Fig. 1, namely in the assignment of the $y$ variable, which will be addressed in the future. Another general limitation of the software arises from the recursion limits imposed by Python, which can be addressed by extending the limit, however, it is not recommended. This can be solved in the future by replacing recursivity with a stack.

## An illustration

We illustrate the application of the `idDL2dDL` tool with the analysis of a piecewise linear model of a biological regulatory network [3].

Biological regulatory networks are complex systems describing biological phenomena such as cell metabolism. This kind of model describes the physical and chemical interaction between cell proteins, mRNA, genes, and other cell organelles. When applying a deterministic model to such system, the formalism which is considered to accurately describe such dynamics is a system of nonlinear differential equations. Numerical methods, such as simulations, are then applied to study the complex behavior and interactions between the components of a cell. To fully understand the major dynamics of a biological process, these systems of differential equations often are subjected to a preliminary study. They are firstly simplified by proper methods, resulting in simpler models such as *PieceWise Linear* (PWL) models. This kind of model preserves the major dynamics of the original one and is easier to study . They are used to find the main dynamics of a biological system (such as the existence of attractors or cyclic behaviors) and a detailed study of the identified features comes afterward. A PWL model is composed of several domains containing a system of linear differential equations which are obtained by proper simplifications of the (nonlinear) original one (check [3] for details).

**Example 2** *In [5] we can found an example of a PWL model.*

| | | |
|---|---|---|
| $\begin{cases} x' = -x \\ y' = -y \end{cases}$ <br> $x < 2$ <br> $2 < y$ | $\begin{cases} x' = -x \\ y' = -y \end{cases}$ <br> $2 < x < 4$ <br> $2 < y$ | $\begin{cases} x' = -x \\ y' = 3 - y \end{cases}$ <br> $4 < x$ <br> $2 < y$ |
| $\begin{cases} x' = -x \\ y' = -y \end{cases}$ <br> $x < 2$ <br> $y < 2$ | $\begin{cases} x' = 5 - x \\ y' = -y \end{cases}$ <br> $2 < x < 4$ <br> $0 < y < 2$ | $\begin{cases} x' = 5 - x \\ y' = 3 - y \end{cases}$ <br> $4 < x$ <br> $0 < y < 2$ |

This system is characterized by having continuous dynamics within each domain but discrete reconfigurations when we move from one domain to another. Continuous variables describe the concentration of intracellular components, such as proteins and RNA. This hybrid dynamics can be expressed by a hybrid program of $d\mathscr{L}$.

For instance, the behaviour of the system running in the the $(0,0)$-square (i.e. the left-bot one), is expressed as the hybrid program $bio_{00} \equiv ?(x < 2 \wedge y < 0); (x' = -x, y' = -y \& (x < 2 \wedge y < 0))$ where the part $?(x < 2 \wedge y < 0)$ assures the precondition that the system is in fact in a state where $x < 2 \wedge y < 0$; that guards the continuous evolution of $x$ and $y$ according the differential equations $x' = -x, y' = -y$, whenever the condition $x < 2 \wedge y < 0$ is not violated.

Making the same for the other five domains (and aggregating $bio_{00}$ and $bio_{01}$ in $bio_0$), we have:

$bio_0 \equiv ?(x < 2); (x' = -x, y' = -y \& (x < 2))$
$bio_{10} \equiv ?(2 < x \wedge x < 4 \wedge 0 < y \wedge y < 2); (x' = 5 - x, y' = -y \& (2 < x \wedge x < 4 \wedge 0 < y \wedge y < 2))$
$bio_{20} \equiv ?(4 < x \wedge 0 < y \wedge y < 2); (x' = 5 - x, y' = 3 - y \& (4 < x \wedge 0 < y \wedge y < 2))$
$bio_{11} \equiv ?(2 < x \wedge x < 4 \wedge 2 < y); (x' = -x, y' = -y \& (2 < x \wedge x < 4 \wedge 2 < y))$
$bio_{21} \equiv ?(4 < x \wedge 2 < y); (x' = -x, y' = 3 - y \& (4 < x \wedge 2 < y))$

We can obtain the hybrid program describing the dynamics of the entire biological system:

$$bio \equiv (bio_0 \bigcup bio_{10} \bigcup bio_{20} \bigcup bio_{11} \bigcup bio_{21})^*$$

Then we can take advantage of the interval syntax to express biological properties like "when the concentrations $x$ and $y$ are around 5.5 and 3.5, the biological system will never reach a state where $x < 2$".

$$\left[x := [5,6]\,; y := [3,4]\right] \left[bio\right] x > 2$$

This example was then translated and proven in `KeYmaera X`, using the default automatic proof tactic,


Figure 3: The translation of the biological regulatory network example in `idDL2dDL` and the respective proof in `KeYmaera X`.

according to Fig. 3.

Being representative of scenarios treatable with the framework, the illustration considered dealt with a structure PWL containing a system of linear differential equations within each domain and a verification property particularly well behaved, in the sense that the engine was able to close the proof automatically. We note however that `KeYmaera X` is semi-automatic prover and, hence it often needs human intervention to aid on the proofs. Naturally, in practices, a system comprising additional variables and a property

described by a more complex formula will certanly require human intervention to assists the verification task.

## 3 Discussion and conclusion

This paper introduces `idDL2DL`, a parser and translator from interval dynamic logic formulas to $d\mathscr{L}$. An application of the tool was illustrated with a case study of a biological regulatory network model, that was translated to $d\mathscr{L}$ syntax and analyzed in `KeyMaera X`.

As aforementioned, more than a new logic, the formalism introduced here represents an adaptation of $d\mathscr{L}$ driven to specific applications, namely in the synthetic biology domain. It consists of a syntax directed to interval contexts along with adapted semantics, to inherit the soundness from $d\mathscr{L}$ (see [10, 4]. We notice that interval arithmetic has already been considered in $d\mathscr{L}$, through a different approach, in [1]. In that work, for instance, a third truth-value $U$ is considered for uncertain statements like $[0,2] < [1,3]$. These kinds of propositions are evaluated as $false$ in the present work, to carry a conservative approach. Consequently, our semantical interpretation of continuous evolutions was adapted, not being so restrictive to catch every punctual $n$-dimension initial state. This allows `KeyMaera X` to consider every possible continuous evolution as in $d\mathscr{L}$ and, in this way, preserve the soundness (see [4] for details).

This tool still has room for multiple improvements, mainly when it comes to extending the pre-processing capabilities of the interpreter. With this user-friendly interface, we aim to open the tool for users without experience in formal verification of systems, such as researchers from application areas, as is the case of synthetic biology.

## References

[1] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen & André Platzer (2018): *VeriPhy: Verified Controller Executables from Verified Cyber-Physical System Models*. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Association for Computing Machinery, p. 617–630.

[2] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen & André Platzer (2018): *VeriPhy: Verified Controller Executables from Verified Cyber-Physical System Models*. *SIGPLAN Not.* 53(4), p. 617–630, doi:10.1145/3296979.3192406. Available at `https://doi.org/10.1145/3296979.3192406`.

[3] Hidde De Jong (2002): *Modeling and simulation of genetic regulatory systems: a literature review*. *Journal of computational biology* 9(1), pp. 67–103.

[4] Daniel Figueiredo (2021): *Introducing interval differential dynamic logic*. In Hossein Hojjat & Mieke Massink, editors: *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods*, *Lecture Notes in Computer Science* 12818, Springer, pp. 69–75.

[5] Daniel Figueiredo & Luís Soares Barbosa (2018): *Reactive Models for Biological Regulatory Networks*. In Madalena Chaves & Manuel A. Martins, editors: *Molecular Logic and Computational Synthetic Biology, MLCSB 2018*, *Lecture Notes in Computer Science* 11415, Springer, pp. 74–88.

[6] Daniel Figueiredo, Manuel A. Martins & Madalena Chaves (2017): *Applying differential dynamic logic to reconfigurable biological networks*. *Mathematical Biosciences* 291, pp. 10 – 20.

[7] David Harel, Jerzy Tiuryn & Dexter Kozen (2000): *Dynamic Logic*. MIT Press, Cambridge, MA, USA.

[8] Ramon E. Moore (1962): *Interval Arithmetic and Automatic Error Analysis in Digital Computing*. Ph.D. thesis, Stanford University.

[9] Andreas Müller, Stefan Mitsch, Wieland Schwinger & André Platzer (2018): *A Component-Based Hybrid Systems Verification and Implementation Tool in KeYmaera X (Tool Demonstration)*. In Roger D. Chamberlain, Walid Taha & Martin Törngren, editors: *Cyber Physical Systems. Model-Based Design, CyPhy 2018*, *Lecture Notes in Computer Science* 11615, Springer, pp. 91–110.

[10] André Platzer (2018): *Logical Foundations of Cyber-Physical Systems*. Springer.

[11] Ricardo G. Sanfelice, David A. Copp & Pablo Nanez (2013): *A toolbox for simulation of hybrid systems in matlab/simulink: hybrid equations (HyEQ) toolbox*. In Calin Belta & Franjo Ivancic, editors: *Proceedings of the 16th international conference on Hybrid systems:computation and control, HSCC 2013*, ACM, pp. 101–106.

[12] Regivan H. N. Santiago, Benjamín R. C. Bedregal, Alexandre Madeira & Manuel A. Martins (2019): *On interval dynamic logic: Introducing quasi-action lattices*. *Sci. Comput. Program.* 175, pp. 1–16, doi:10.1016/j.scico.2019.01.007. Available at `https://doi.org/10.1016/j.scico.2019.01.007`.

# A Formal Proof of the Strong Normalization Theorem for System T in Agda

Sebastián Urciuoli*
Universidad ORT Uruguay
Montevideo, Uruguay
urciuoli@ort.edu.uy

We present a framework for the formal meta-theory of lambda calculi in first-order syntax, with two sort of names, one to represent both free and bound variables, and the other for constants, and using Stoughton's multiple substitutions. On top of the framework we formalize Girard's proof of the Strong Normalization Theorem for both the simply-typed lambda calculus and System T. As to the latter, we also present a simplification of the original proof. The whole development has been machine-checked using the Agda system.

## 1 Introduction

In [21] a framework was presented for the formal meta-theory of the pure untyped lambda calculus in first-order abstract syntax (FOAS) and using only one sort of names for both free and bound variables[1]. Based upon Stoughton's work on multiple substitutions [18], the authors were able to give a primitive recursive definition of the operation of substitution which does not identify alpha-convertible terms[2], avoids variable capture, and has a homogeneous treatment in the case of abstractions. Such a definition of substitution is obtained by renaming every bound name to a sufficiently fresh one. The whole development has been formalized using the Agda system [15].

The framework has been used since then to verify many fundamental meta-theoretic properties of the lambda calculus including: Subject Reduction for the simply-typed lambda calculus (STLC) in [6]; The Church-Rosser Theorem for the untyped lambda calculus also in [6]; The Standardization Theorem in [7], and; The Strong Normalization Theorem for STLC in [24], and by using F. Joachimski and R. Matthes' syntactical method [12]. Now in this paper we continue the same line of work, and formalize the Strong Normalization Theorem for System T, and we also present a new and different mechanization for STLC.

System T extends STLC by adding primitive recursive functions on natural numbers. It has its roots in K. Gödel's work presented in [10], and it was originally developed to study the consistency of Peano arithmetic. The Strong Normalization Theorem states that every program (term) in some calculus under consideration is strongly normalizing. A term is *strongly normalizing* if and only if its computation always halts regardless of the reduction path been taken. This result for System T is already well known. In this development we mechanize J.-Y. Girard's proof presented in [9], which in turn is based on W. W. Tait's method of *computability* or *reducible functions* [20] (henceforth we shall refer to Girard and Tait's method or proof interchangeably). This method defines a (logical) relation between terms

*This work is partially supported by Agencia Nacional de Investigación e Innovación (ANII), Uruguay.

[1]Both the previous framework and the one presented here use named variables, it bears repeating. In a contrary sense, there are nameless approaches, e.g., de-Bruijn indices [4] or locally nameless syntax [5], which use numbers to identify the variables.

[2]Or without using Barendregt's variable convention.

and types that is fitter than the Strong Normalization Theorem, and hence it enables a more powerful induction hypothesis. Any term related to some type under such a relation is said to be *reducible*. Then the method consists of two steps: first, to prove that all reducible term are strongly normalizing, and secondly to prove that all typed terms are reducible.

Initially, the sole objective of this work was to formalize a proof of the Strong Normalization Theorem but only for System T, and by using the framework presented in [21]. Of course, the syntax of the pure lambda terms had to be extended to include the term-formers for the natural numbers and the recursion operator[3]. For this, we based ourselves upon a standard definition of the lambda terms in which two disjoint sort of names are used, one to represent the variables, and the other for the constants, e.g., see [11]. Now, instead of restricting ourselves to a specific set of constants, we shall allow any (countable) set. Once the syntax of the framework had been parameterised it felt natural to parameterise the reduction schema as well, as these relations are often defined by the syntax. The work went a bit further, and the first part of the proof was also abstracted for a class of calculi to be defined; this step consists mainly in analysing reduction paths. To round up, hitherto the work evolved from formalizing the proof of the Strong Normalization Theorem in System T, into also providing a general-purpose framework with theories for substitution, alpha-conversion, reduction and reducible terms of simple types.

Now, having such a framework it was a good time to revisit the previous formalization of the Strong Normalization Theorem for STLC presented in [24]. There, the definition of the logical relation was based on the one in the POPLmark Challenge 2 [1], and it included the context of variables. In addition to that, a syntactical characterization based on [12] was used to define the type of the strongly normalizing terms. In this development, we shall use a standard definition of the logical relation which does not contain the context, and an accessibility characterization of the strongly normalizing terms based on [2]. Furthermore, the proof for STLC is contained in the one for System T, so it serves both as a milestone in this exposition, as well as to show the incremental nature of the whole method presented here.

The last result presented in this development is about a simplification in Girard's proof of the Strong Normalization Theorem for System T. More specifically, in the second part of the proof there is a lemma whose principle of induction requires to count the occurrences of the successor operator in the *normal form* of a given strongly normalizing term. This is not strictly necessary, and one can just count such symbols *directly* in the term, and so *avoid evaluating* it.

In summary, the novel contributions in this paper are: (1) a framework for the meta-theory of lambda calculi in FOAS with named variables and constants; (2) a complete mechanization of Girard's proof of The Strong Normalization Theorem for System T in Agda; (3) a new and different mechanization of Girard's proof for STLC in Agda as well, and; (4) a simplification on the principle of induction of Girard's original proof of The Strong Normalization Theorem for System T. To the best of our knowledge, there is not yet a formalization of the Strong Normalization Theorem for System T. The development has been entirely written in Agda and it is available at: `https://github.com/surciuoli/lambda-c`.

The structure of this paper is the following. In the next section we introduce the new framework: its syntax, substitution, conversion theories and logical relations (reducible terms). Some results presented are completely new, and some others are an extension of [21, 24] to consider the additional syntax. From Section 2.5 on, and unless the opposite is explicitly stated, all results represent new developments. In Section 3, we formalize both STLC and Girard's proof of the Strong Normalization Theorem. In Section 4, we extend both the calculus and Girard's proof to System T, and we also explain the aforementioned simplification. In the last sections we give some overall conclusions and compare our work with related

[3] In this development we shal not consider booleans nor tuples as part of the syntax. Nevertheless, they can be easily defined by the machinery presented here.

developments. Throughout this exposition we shall use Agda code for definitions and lemmata, and a mix of code and English for the proofs in the hope of making reading more enjoyable.

It is assumed that the reader is familiarized with dependently-typed programming languages, preferably with Agda. Alternatively, the reader might have some background on any of the variants of constructive (or intuitionistic) type theory, e.g., Per-Martin Löf's formulation [13] which is the one upon Agda is founded.

## 2 The Framework

Let $\mathtt{V} = v_0, v_1 \ldots$ be any infinitely countable collection of names, the variables, ranged over by letters $x$, $y \ldots$ and equipped with a deciding procedure for definitional equality; for concreteness, we shall define $\mathtt{V} = \mathbb{N}$, i.e., the set of natural numbers in Agda, but it can be any other suitable type, e.g., strings. Let `C` be any possibly infinite countable collection of names, the constants, and ranged over by $c$. The abstract syntax of the lambda terms with constants is defined:

**Definition 2.1** (Syntax)**.**

```
1 module CFramework.CTerm (C : Set) where
2 ...
3 data Λ : Set where
4   k : C → Λ
5   v : V → Λ
6   ƛ : V → Λ → Λ
7   _·_ : Λ → Λ → Λ
```

In line 1 we indicate that the definition is contained in the module `CFramework.CTerm`, which according to Agda's specification must be located in the file CFramework/CTerm.agda. We also specify that the module is parameterised by the set of constants `C`, which can be of any inductive type (`Set`). Lines 4 and 5 define the constructors for the constants and the variables respectively. In line 6 we use `ƛ` to not interfere with Agda's primitive $\lambda$. We shall follow the next convention unless the opposite is explicitly stated: use $\lambda$ to represent *object-level* abstractions in informal discussions and proofs, and use `ƛ` in code listings. Line 7 defines the infix binary operator of function application. As usual, we shall use letters $M$, $N \ldots$ to range over terms.

The module can be then instantiated with any type of constants. For example, the next declaration derives the syntax of the pure lambda terms into the current scope:

**Definition 2.2.** `open import CFramework.CTerm ⊥`

$\bot$ is the inductive type without any constructor. The `import` statement tells Agda to load the content of the file named after the module into the current scope, while the `open` statement lets one access the definitions in it without having to qualify them. Both statements can be combined into a single one as shown.

Whenever a name $x$ syntactically occurs in a term $M$ and is not bound by any abstraction, we shall say $x$ is free in $M$, and write it $x * M$. On the other hand, if every occurrence of $x$ is bound by some abstraction (or even if $x$ does not occur at all), we shall say $x$ is fresh in $M$, and write it $x \# M$ as in nominal techniques, e.g., see [23]. Both relations are inductively defined in a standard manner, and in [21] it was proven that both relations are opposite to each other.

It will come in handy to define both the type of predicates and binary relations on terms respectively by: `Pred = Λ → Set`, and `Rel = Λ → Λ → Set`.

## 2.1 Substitution

Substitution is the fundamental entity on which alpha- and beta-conversion sit. We shall base ourselves upon the work done in [18], and first define multiple substitutions as functions from variables to terms:

```
Subst = V → Λ
```

We shall use letter $\sigma$ to range over them. Later, by applying these functions to the free variables in a given term we shall obtain the desired operation of the *action of substitution* (Definition 2.6), i.e., the operation of replacing every free name $x$ in $M$ by its corresponding image $\sigma x$.

Most substitutions appearing in properties and definitions are identity-almost-everywhere. We can generate them by starting from the identity substitution $\iota$, which maps every variable to itself, and applying the update operation on substitutions `_≺+_` such that for any $\sigma$, $x$ and $M$, $\sigma \prec + (x\ ,\ M)$ is the substitution that maps $x$ to $M$, and $y$ to $\sigma y$ for every $y$ other than $x$:

**Definition 2.3** (Update operation)**.**

```
1  _≺+_ : Subst → V × Λ → Subst
2  (σ ≺+ (x , M)) y with x ≟ y
3  ... | yes _ = M
4  ... | no  _ = σ y
```

In line 1, `×` is the non-dependent product type, and in line 2, $\stackrel{?}{=}$ is the procedure that decides if two names are equal, and mentioned at the start of this section.

In some places we shall need to restrict the domain of a substitution so to have a finite image or range, therefore we introduce the type of restrictions, written `R`, and defined: `R = Subst × Λ`. Below we extend freshness to restrictions:

**Definition 2.4** (Freshness on restrictions)**.**

```
_#⌊_ : V → R → Set
x #⌊ (σ , M) = (y : V) → y ∗ M → x # σ y
```

In English, a name is fresh in the restriction $(\sigma\ ,\ M)$ if and only if it is fresh in every image $\sigma y$, for every $y * M$.

Now we shall briefly discuss the mechanism in the framework used to rename the bound names in a given term, and so avoid capturing any free variable during the action of substitution. The complete description can be found in [21]. Let `χ'` be the function that returns the first name not in a given list:

```
χ' : List V → V
```

The algorithm is obtained by a direct consequence of the pigeonhole principle: the list of names given is finite, therefore we can always choose a fresh name from the infinite collection `V`. Then we can define the choice function $\chi$ that returns the first name not in a given restriction $(\sigma\ ,\ M)$, by first concatenating into a single list every free name that appears in the image $\sigma x$ for any $x * M$, and then selecting the first name not in such a list by using the previous $\chi'$ function:

```
χ : R → V
χ (σ , M) = χ' (concat (mapL (fv ∘ σ) (fv M)))
```

`mapL` applies a function to every element in a list, $\circ$ stands for the usual composition of functions, and `fv` computes the list of free names in a given term. In [21] it was proven that $\chi$ computes a sufficiently fresh name, according to our expectations to be addressed shortly:

**Lemma 2.5.** `χ-lemma2 : (σ : Subst) (M : Λ) → χ (σ , M) #⌊ (σ , M)`

The action of a substitution $\sigma$ on a term $M$ is the operation that replaces every free name in $M$ by its corresponding image under $\sigma$. It is written $M \bullet \sigma$ and defined:

**Definition 2.6** (Action of substitution)**.**

```
_•_ : Λ → Subst → Λ
k c • σ = k c
v x • σ = σ x
M · N • σ = (M • σ) · (N • σ)
ƛ x M • σ = ƛ y (M • σ ≺+ (x , v y)) where y = χ (σ , ƛ x M)
```

Notice that in the last equation we always rename the bound variable $x$ to $y$ by using the $\chi$ function. We can show that this method avoids variable capture: for any $w * M$ other than $x$ it must follow $y \# (\sigma \prec\!+ (x\ ,\ y))w$, otherwise it would mean that we have captured an *undesired* free occurrence of $y$. Notice that if $w = x$ then its image is $y$ which represents an occurrence of $x$ in the original term $\lambda xM$ and therefore must be "re-bound". So, $x * M$ and $x \neq w$, therefore $w * \lambda xM$. Next, by Lemma 2.5 we have $y \# \lfloor (\sigma\ ,\ \lambda xM)$. Then by Definition 2.4 it follows $y \# \sigma w$, and so $y \# (\sigma \prec\!+ (x\ ,\ y))w$ since $(\sigma \prec\!+ (x\ ,\ y))w = \sigma w$ by Definition 2.3.

Unary substitution is defined:

```
_[_/_] : Λ → Λ → V → Λ
M [ N / x ] = M • ι ≺+ (x , N)
```

Our definition of $\bullet$ has a direct consequence on the terms: when submitted to substitutions the bound variables become "ordered", for the lack of a better name. Consider the next example. Let $M = \lambda v_1 v_1$. By definition $M \bullet \sigma = \lambda x(v_1 \bullet \sigma \prec\!+ (v_1\ ,\ x)) = \lambda xx$, where $x = \chi(\sigma\ ,\ \lambda v_1 v_1)$, and for every $\sigma$. We can see that $M$ does not contain any free variable, therefore by definition of $\chi$ we have that $x = v_0$, i.e., the first name in `V`, and so we have that the closed term $\lambda v_1 v_1$ turned into $\lambda v_0 v_0$ even tough no substitution actually happened. Another example a bit more sophisticated is the next one: $(\lambda v_3 \lambda v_2 \lambda v_0 (v_0 v_1 v_2 v_3))\,[\,v_0 \,/\, v_1\,] = \lambda v_1 \lambda v_2 \lambda v_3 (v_3 v_0 v_2 v_1)$. This collateral effect will have some implications on our definition of beta-reduction.

## 2.2 Alpha-conversion

Alpha-conversion is inductively defined by the syntax:

```
1 module CFramework.CAlpha (C : Set) where
2 open import CFramework.CTerm C
3 ...
4 data _∼α_ : Rel where
5   ∼k : {c : C} → k c ∼α k c
6   ∼v : {x : V} → v x ∼α v x
7   ∼· : {M M' N N' : Λ} → M ∼α M' → N ∼α N' → M · N ∼α M' · N'
8   ∼ƛ : {M M' : Λ} {x x' y : V} → y # ƛ x M → y # ƛ x' M'
9      → M [ v y / x ] ∼α M' [ v y / x' ] → ƛ x M ∼α ƛ x' M'
```

Since the syntax of the lambda terms is parameterised by a set `C`, every module that depends on the syntax (all of them) will have to be parameterised by `C` as well. Lines 1 and 2 illustrate this point.

Arguments written between braces `{` and `}` are called *implicit* and they are not required to be supplied; the type-checker will infer their values, whenever possible. Implicit arguments can be made explicit by

enclosing them between braces, e.g., $\sim$`k {c₁}` has type `k c₁` $\sim\alpha$ `k c₁`. For a more detailed explanation on this topic the reader may refer to [19].

The only case in the definition worth mentioning is $\sim$ƛ. There, we rename both $x$ and $x'$ to a common fresh name $y$. If such results are alpha-convertible, then the choice of the bound name is irrelevant, and it should be expected to assert that both abstractions are alpha-convertible. This definition can also be seen in nominal techniques, e.g., see [23], though there it happens to be more usual to rename only one side of $\sim\alpha$. Our symmetrical definition has some advantages over those that are not (see [21]). Also, in [21], $\sim\alpha$ was proven to be an equivalence relation.

The next results are quickly extended from [21]:

**Lemma 2.7.** `lemma•ι : ∀ {M} → M` $\sim\alpha$ `M • ι`

**Lemma 2.8.** `corollary1SubstLemma : ∀ {x y σ M N} → y #↓ (σ , ƛ x M) → (M • σ ≺+ (x , v y)) • ι ≺+ (y , N)` $\sim\alpha$ `M • σ ≺+ (x , N)`

Arguments preceded by ∀ are not required to be annotated with their respective types.

## 2.3 Reduction

Let $\triangleright$ be any binary relation on terms and called a contraction relation. The syntactic closure of $\triangleright$ is written $\rightsquigarrow$ and it is inductively defined:

**Definition 2.9.**

```
1 import CFramework.CTerm as CTerm
2 module CFramework.CReduction (C : Set) (_▷_ : CTerm.Rel C) where
3 open CTerm C
4 ...
5 data _⇝_ : Rel where
6   abs : ∀ {x M N} → M ⇝ N → ƛ x M ⇝ ƛ x N
7   appL : ∀ {M N P} → M ⇝ N → M · P ⇝ N · P
8   appR : ∀ {M N P} → M ⇝ N → P · M ⇝ P · N
9   redex : ∀ {M N} → M ▷ N → M ⇝ N
```

Line 1 imports the module `CFramework.CTerm`, and at the same time renames it to `CTerm` just for convenience. Line 2 specifies that the module is parameterised by the contraction relation $\triangleright$; notice that since we have neither opened the module `CTerm` nor specified the set of constants to be used, we wrote `CTerm.Rel C` (compare with line 5). From now until the end of this section, it is assumed that both `C` and $\triangleright$ are in the scope of every definition unless explicitly stated the opposite.

Any term on the left-hand side of $\triangleright$ shall be called a redex, as usual, and any term on right-hand side a contractum. Besides, any term on the right-hand side of $\rightsquigarrow$ shall be called a reductum.

We can define beta-reduction by means of $\rightsquigarrow$ as next. Let beta-contraction be inductively defined:

**Definition 2.10** (Beta-contraction)**.**

```
module CFramework.CBetaContraction (C : Set) where
...
data _▷β_ : Rel where
  beta : ∀ {x M N} → ƛ x M · N ▷β M [ N / x ]
```

Then beta-reduction for the pure lambda calculus is derived by importing the modules:

**Definition 2.11** (Beta-reduction)**.**

```
open import CFramework.CBetaContraction ⊥
open import CFramework.CReduction ⊥ _▷β_ renaming (_⇝_ to _↠β_)
```

Recall that in Definition 2.2 we had explained that by defining $\mathtt{C} = \bot$ we obtain the syntax of the pure lambda terms.

The renaming done by $\bullet$ is sensitive to the free variables in the subject term. As a consequence, $\twoheadrightarrow\beta$ is not compatible with substitution, i.e., the next lemma *does not* hold:

```
∀ {M N σ} → M ↠β N → M • σ ↠β N • σ
```

Consider the following example. Let $M = \lambda v_1((\lambda v_0 \lambda v_0 v_0)v_0)$ and $N = \lambda v_1 \lambda v_0 v_0$. It can be seen that $M \twoheadrightarrow\beta\ N$ is derivable. Now, let us apply $\iota$ on each side. As to $N$, $v_1$ is renamed to the first name fresh in the restriction $(\iota\ ,\ \lambda v_1 \lambda v_0 v_0)$, i.e., to $v_0$; we obtain $N \bullet \iota = \lambda v_0 \lambda v_0 v_0$. As to $M$, the variable $v_1$ is renamed to itself, since it is the first fresh name in the corresponding restriction (renaming it to $v_0$ would cause a capture). So, $M \bullet \iota = M$, and the only reductum $\lambda v_1((\lambda v_0 v_0)[\ v_0\ /\ v_0\ ])$ of $M$ equals to $\lambda v_1 \lambda v_0 v_0$, which is not $N \bullet \iota$.

Since we are going to need some form of the lemma of compatibility above as we shall see, we will use the next approximation which is always possible: continuing with the earlier example, after the reduction takes place we shall perform an alpha-conversion step from the reductum to meet $N \bullet \iota$, i.e., $M \bullet \iota \twoheadrightarrow\beta\ \lambda v_1 \lambda v_0 v_0$ followed by $\lambda v_1 \lambda v_0 v_0 \sim\alpha N \bullet \iota$.

So, let $r$ be any binary relation on terms, either a contraction relation or a reduction. We shall say $r$ is alpha-compatible with substitution, and write it $\mathtt{Compat}\bullet\ r$, if and only if, for every directed pair of terms $M$ and $N$ related under $r$ both there must exist some $P$ such that $M \bullet \sigma$ and $P$ are also related, and $P \sim\alpha N \bullet \sigma$. Formally:

**Definition 2.12** (Alpha-compatibility with substitution)**.**

```
Compat• r = ∀ {M N σ} → r M N → Σ[ P ∈ Λ ](r (M • σ) P × P ∼α N • σ)
```

In Agda, the dependent product type can be written `Σ[ a ∈ A ] B`, where `a` is some (meta-)variable of type `A`, and `B` is some type which might depend upon `a`.

Similarly, we shall say $r$ is alpha-commutative and define it:

**Definition 2.13** (Alpha-commutativity)**.**

```
Comm∼α r = ∀ {M N P} → M ∼α N → r N P → Σ[ Q ∈ Λ ](r M Q × Q ∼α P)
```

We shall restrict this development to contraction relations that preserve freshness, i.e., that do not introduce any free name in any contractum:

**Definition 2.14.** `Preserves# r = ∀ {x M N} → r M N → x # M → x # N`

Then we have that, if $\triangleright$ preserves freshness, or it is compatible with substitution, or it commutes with alpha-conversion, then its syntactic closure has the corresponding properties as well:

**Lemma 2.15.**

```
preser⇝# : Preserves# _▷_ → Preserves# (_⇝_ _▷_)
compat⇝• : Preserves# _▷_ → Compat• _▷_ → Compat• (_⇝_ _▷_)
commut⇝α : Preserves# _▷_ → Compat• _▷_ → Comm∼α _▷_ → Comm∼α (_⇝_ _▷_)
```

Their proofs are extended from [21, 24]. Notice the cascade effect on the lemmata: each of them has all the arguments of the one above. This happens naturally since each lemma relies on the previous one.

Finally, we have that beta-contraction is alpha-commutative, along with two other results (their proofs are extended from [24]):

**Lemma 2.16.** `Preserves# _▷β_ × Compat• _▷β_ × Comm∼α _▷β_`

## 2.4 Strongly normalizing terms

A term is strongly normalizing if and only if every reduction path starting from it eventually halts. We shall use their accessible characterization (originally presented in [2]). For any given computation relation $\rightsquigarrow$ we define sn:

**Definition 2.17** (Strongly normalizing terms)**.**

```
1 sn : Λ → Set
2 sn = Acc (dual _⇝_)
```

Acc is the type of the accessible elements by some order $<$, i.e., the set of elements $a$ such that there is no infinite sequence $\ldots < a' < a$. It is defined in Agda's standard library [22]. dual is the function that returns the *type* of the inverse of every binary relation on terms. We use the dual of $\rightsquigarrow$ instead of the direct because Acc expects an order that descends to its left-hand side, so to speak, which is not the case for $\rightsquigarrow$. Line 2 can be read as: sn is the set of terms $M$ such that $M \rightsquigarrow M' \rightsquigarrow \ldots$ is always finite. Below is the definition of Acc to support this paragraph:

```
data Acc {a b} {A : Set a} (_<_ : Rel A b) (x : A) : Set (a ⊔ b) where
  acc : (∀ y → y < x → Acc _<_ y) → Acc _<_ x
```

Note that Rel above is the type of binary relations between any two types, and it is defined in the standard library as well.

The next result is adapted from [24] and follows easily by induction:

**Lemma 2.18.** `inversionSnApp : ∀ {M N} → sn (M · N) → sn M × sn N`

sn is closed under alpha-conversion, as long as the supporting relation $\rightsquigarrow$ commutes with alpha-conversion. The corresponding proof presented here is an adaptation of [24]:

**Lemma 2.19.** `closureSn∼α : Comm∼α _⇝_ → ∀ {M N} → sn M → M ∼α N → sn N`

*Proof.* By induction on the derivation of $\mathtt{sn}M$. To derive $\mathtt{sn}N$ we need to prove $\mathtt{sn}P$ for any $N \rightsquigarrow P$. By Definition 2.13 there exists some $Q$ such that $M \rightsquigarrow Q$ and $Q \sim\alpha P$. By Definition 2.17, $\mathtt{sn}Q$ holds, i.e., $Q$ is accessible, and $\mathtt{sn}Q$ is a proper component of the derivation of $\mathtt{sn}P$[4]. Then, we can use the induction hypothesis on $\mathtt{sn}Q$ together with $Q \sim\alpha P$ and obtain $\mathtt{sn}P$. □

Exceptionally we show the code of the proof above because it is very compact, and to reinforce the understanding of the structural principle of induction of sn:

```
closureSn∼α comm {M} {N} (acc i) M∼N =
  acc λ P P↞N → let Q , M↠Q , Q∼P = comm M∼N P↞N
                in closureSn∼α comm (i Q M↠Q) Q∼P
```

The $\lambda$ occurrence denotes Agda's entity for meta-level lambda terms. `i Q M↠Q` is of type $\mathtt{sn}Q$, and it is a proper component of `acc i` which is of type $\mathtt{sn}M$. `P↞N` is of type `(dual _⇝_) P N` which in turn equals to `P ⇝ N`. For the same reason `M↠Q` is of type `(dual _⇝_) Q M`.

[4] Put in other words, every reduction beginning in $Q$ is at least one step shorter than every other reduction beginning in $M$.

## 2.5 Reducibile terms

Girard's proof of the Strong Normalization Theorem defines a relation between terms and types. A term that is related to some type is said to be reducible. The proof is carried out in two steps: first, it is proven that every reducible term is strongly normalizing, and secondly that every typed term is reducible. In this section we shall define the logical relation of reducible terms, and after that we shall prove some of their properties, including the first step in Girard's proof (`CR1` of Lemma 2.24).

Both in STLC and System T (object-level) types are simple, so regarding this development they will be enough for our definition of the logical relation. We define them by:

```
data Type : Set where
  τ : Type
  _⇒_ : Type → Type → Type
```

Then, the relation of reducible terms or logical relation is defined by recursion on the types:

**Definition 2.20** (Reducible terms)**.**

```
Red : Type → Λ → Set
Red τ M = sn M
Red (α ⇒ β) M = ∀ {N} → Red α N → Red β (M · N)
```

`Red` is also closed under alpha-conversion:

**Lemma 2.21** (Closure of `Red` under $\sim\alpha$)**.**

```
closureRed∼α : Comm∼α _⇝_ → ∀ {α M N} → Red α M → M ∼α N → Red α N
```

*Proof.* By induction on the type $\alpha$, and by using Lemma 2.19. □

Next we have neutral terms. We shall use a different characterization than the one given in [9], and define them as the set of terms that contains all variables, in addition to all terms which when iteratively applied to any sequence of terms, the result is never a redex, i.e., if $M$ is neutral then $MN_0N_1\ldots N_n$ is not a redex for any $n > 0$. Since we are defining a general-purpose framework, we need to find an abstract characterization wide enough to hold for the targeted calculi. The next definition was found to be appropriate:

**Definition 2.22** (Neutral terms)**.**

```
1 record ConditionsNe (Ne : Pred) : Set where
2   field cond1 : ∀ {x} → Ne (v x)
3         cond2 : ∀ {M} → Ne M → ∀ {N} → Ne (M · N)
4         cond3 : ∀ {M} → Ne M → ∀ {N P} → ¬((M · N) ▷ P)
```

In line 1, the parameter on the left-hand side of the last occurrence of `:` indicates that `Ne` is an argument constant to every field or constructor (similar to what happens with modules). In line 4, `¬ A` is the negation of any proposition (or type) `A`, and it is defined by: `¬ A = A → ⊥`. Finally, `Ne`$M$ shall be read as: *$M$ is neutral*.

We have packed the conditions that identify any predicate as a possible definiens for the set of neutral terms into a single record just for a matter of taste. In Agda, a record is a $n$-tuple with tags or labels for accessing each element. Any object of type `ConditionsNe Ne` will then be a proof of that such a predicate `Ne` is a good candidate to represent the type of neutral terms, i.e., `Ne`$x$ holds for every $x$ (`cond1`), and for every `Ne`$M$ it follows both that `Ne`$(MN)$ (`cond2`), and that $MN$ is not a redex (`cond3`), which combined these last two statements ensure that no application $MN_0N_1\ldots$ is ever a redex for any `Ne`$M$.

Finally, we shall need to require that no variable be a redex in $\triangleright$[5]:

[5] Actually, we only need one variable to not be a redex (see the proof of `CR1` in Lemma 2.24, when $\alpha$ is functional).

**Definition 2.23** (Condition of the contraction relation)**.**

```
record Conditions▷ : Set where
  field cond4 : ∀ {x M} → ¬(v x ▷ M)
```

We used a record just to maintain coherence with previous conditions.

As to the main result in this section, we have the next properties about reducible terms, where `CR1` corresponds to the first part of Girard's proof:

**Lemma 2.24** (Properties of reducible terms)**.**

```
module RedProperties (Ne : Pred) (p : ConditionsNe Ne) (q : Conditions▷) where
...
CR1 : ∀ {α M} → Red α M → sn M
CR2 : ∀ {α M N} → Red α M → M ⇝ N → Red α N
CR3 : ∀ {α M} → Ne M → (∀ {N} → M ⇝ N → Red α N) → Red α M
```

*Proof.* By mutual induction on the type $\alpha$:

- Case $\alpha = \tau$:
  - `CR1` By Definition 2.20, $\texttt{Red}\,\tau\,M = \texttt{sn}\,M$, so `CR1` is a tautology.
  - `CR2` Immediate by Definition 2.17.
  - `CR3` Analogous to `CR2`.
- Case $\alpha = \beta \Rightarrow \gamma$:
  - `CR1` By `cond1` of Definition 2.22 we have $\texttt{Ne}\,v_0$. By `cond4` of Definition 2.23 together with Definition 2.9 we have that $v_0 \rightsquigarrow N$ is absurd for any $N$, therefore the second hypothesis of `CR3` follows by vacuity, and so we can use the main induction hypothesis `CR3` and obtain $\texttt{Red}\,\beta\,v_0$. Now, by Definition 2.20 on $\texttt{Red}\,(\beta \Rightarrow \gamma)\,M$, we obtain $\texttt{Red}\,\gamma\,(Mv_0)$, and by the induction hypothesis $\texttt{sn}\,(Mv_0)$. Finally, by Lemma 2.18 we get $\texttt{sn}\,M$.
  - `CR2` According to Definition 2.20, to prove the thesis $\texttt{Red}\,(\beta \Rightarrow \gamma)\,N$ we have to prove $\texttt{Red}\,\gamma\,(NP)$ for any $\texttt{Red}\,\beta\,P$. By hypothesis we know $\texttt{Red}\,\gamma\,(MP)$, and by compatibility of $\rightsquigarrow$ with the syntax together with the hypothesis $M \rightsquigarrow N$ we have $MP \rightsquigarrow NP$, and so we can use the induction hypothesis and obtain $\texttt{Red}\,\gamma\,(NP)$ as desired.
  - `CR3` Let $\texttt{Red}\,\beta\,P$. To derive our desired result $\texttt{Red}\,\gamma\,(MP)$ and by using the induction hypothesis, we need to feed it with the required hypotheses or arguments: (1) $\texttt{Ne}\,(MP)$, and (2) that for every $N'$, $MP \rightsquigarrow N'$ implies $\texttt{Red}\,\gamma N'$. (1) follows by `cond2` of Definition 2.22. As to (2), first of all, by the main induction hypothesis `CR1` we get $\texttt{sn}\,P$. Now, we shall continue by a nested induction on the derivation of $\texttt{sn}\,P$[6]. Let us analyse every possible derivation of $MP \rightsquigarrow N'$.
    - Case `redex`: $MP \triangleright N'$ is absurd by Definition 2.23.
    - Case `appL`: If $MP \rightsquigarrow M'N''$ follows from $M \rightsquigarrow M'$ with $N' = M'N''$ then by (2) we get $\texttt{Red}\,(\beta \Rightarrow \gamma)\,M'$, and so by Definition 2.20, $\texttt{Red}\,\gamma\,(M'N'')$.
    - Case `appR`: If $MP \rightsquigarrow MP'$ follows from $P \rightsquigarrow P'$ with $N' = MP'$, then by Definition 2.17 we obtain $\texttt{sn}\,P'$, which is a proper component of $\texttt{sn}\,P$, and so we can continue by induction on $\texttt{sn}\,P'$.

□

[6] In the code, it means to define an auxiliary function in the current scope.

Next we have some general definitions regarding the assignment of types. First, there are contexts (of variable declarations). They are defined as list of pairs, possibly with duplicates:

**Definition 2.25.** `Cxt = List (V × Type)`

Then there is the relation of membership between variables and contexts. We shall write $x \in \Gamma$ and say that $x$ is the *first* variable in $\Gamma$, searched from left to right. Below is the inductive definition:

```
data _∈_ : V → Cxt → Set where
  here  : ∀ {x α Γ} → x ∈ Γ ⩔ x : α
  there : ∀ {x y α Γ} → x ≢ y → x ∈ Γ → x ∈ Γ ⩔ y : α
```

`Γ ⩔ x : α` is syntax-sugar for `(x , α) :: Γ`. Finally, there is a lookup function on contexts such that it returns the type of the first variable (provided it is declared), searched in the same fashion, and defined:

```
1 _⟨_⟩ : ∀ {x} → (Γ : Cxt) → x ∈ Γ → Type
2 []              ⟨ ()       ⟩
3 ((k , d) :: xs) ⟨ here     ⟩ = d
4 ((k , d) :: xs) ⟨ there _ p ⟩ = xs ⟨ p ⟩
```

In the second line, `()` is an *absurd* pattern, and it tells Agda to check that there is no possible way of having an object of type $x \in$ `[]`, for any $x$.

To end this section, we present reducible substitutions. We shall say a substitution is reducible under some context $\Gamma$ if and only if it maps every variable in $\Gamma$ to a reducible term of the same type:

**Definition 2.26.** `RedSubst σ Γ = ∀ x → (k : x ∈ Γ) → Red (Γ ⟨ k ⟩) (σ x)`

The next results follow immediately by definition:

**Lemma 2.27.** `Red-ι : ∀ {Γ} → RedSubst ι Γ`

**Lemma 2.28.**

`Red-upd : RedSubst σ Γ → ∀ x → Red α N → RedSubst (σ ≺+ (x , N)) (Γ ⩔ x : α)`

# 3 STLC

The syntax and theories of substitution, alpha- and beta-reduction for STLC are obtained by instantiating the framework with:

```
module STLC where
open import CFramework.CTerm ⊥
...
open import CFramework.CReduction ⊥ _▷β_ as Reduction renaming (_⇝_ to _→β_)
```

Next is the assignment of types in STLC:

```
data _⊢_:_ (Γ : Cxt) : Λ → Type → Set where
  ⊢var : ∀ {x} → (k : x ∈ Γ) → Γ ⊢ v x : Γ ⟨ k ⟩
  ⊢abs : ∀ {x M α β} → Γ ⩔ x : α ⊢ M : β → Γ ⊢ ƛ x M : α ⇒ β
  ⊢app : ∀ {M N α β} → Γ ⊢ M : α ⇒ β → Γ ⊢ N : α → Γ ⊢ M · N : β
```

### 3.1 The Strong Normalization Theorem in STLC

Following Girard's proof, first we need to prove that every reducible term is sn. We shall use CR1 of Lemma 2.24 for that matter. So, to begin with, we need to give a proper definition of the neutral terms in STLC. Let them be inductively defined by:

```
data Neβ : Pred where
  var : ∀ {x} → Neβ (v x)
  app : ∀ {M N} → Neβ (M · N)
```

One should convince oneself that Neβ respects conditions in Definition 2.22:

**Lemma 3.1.** `conditionsNeβ : Conditions Neβ`

In addition to that, it is also easy to verify that $\triangleright\beta$ (Definition 2.10) does not reduce variables:

**Lemma 3.2.** `conditions▷β : Conditions▷ _▷β_`

Therefore we inherit Lemma 2.24 for STLC by:

```
open import CFramework.CReducibility ⊥ _▷β_ as Reducibility
open Reducibility.RedProperties Neβ conditionsNeβ conditions▷β
```

Now we have to prove that every typed terms is reducible; we shall refer to this as the main lemma. To present the proof, we are going to need some preparatory results. First, by Lemma 2.16 together with Lemma 2.15 we have that $\twoheadrightarrow\beta$ is both alpha-compatible with substitution, and alpha-commutative:

**Lemma 3.3.** `Compat• _↠β_ × Comm∼α _↠β_`

Secondly, since the main lemma proceeds by induction on the derivation of the typing judgement, and the case of abstractions is quite complex, it turns out to be convenient to have a separate lemma for this case:

**Lemma 3.4.** `lemmaAbs : ∀ {x M N α β} → sn M → sn N`

`→ (∀ {P} → Red α P → Red β (M [ P / x ])) → Red α N → Red β (ƛ x M · N)`

*Proof.* By induction on the derivations of sn$M$ and sn$N$. We shall refer to hypotheses sn$M$, sn$N$, $\forall\{P\} \to \mathtt{Red}\,\alpha\,P \to \mathtt{Red}\,\beta\,(M[P\,/\,x])$ and Red$\alpha N$ as (1) through (4) respectively. So, to use CR3 of Lemma 2.24 to prove that the *neutral term* $(\lambda xM)N$ is reducible of type $\beta$ (the thesis of this lemma) we need to show that every reductum is reducible (the second explicit hypothesis of the mentioned lemma). So, let us analyze every possible case:

- Case redex: If $(\lambda xM)N \twoheadrightarrow\beta\ M[N\,/\,x]$ then we can quickly derive that $M[N\,/\,x]$ is reducible from (3) and (4).
- Case appL: If $(\lambda xM)N \twoheadrightarrow\beta\ (\lambda xM')N$ follows from $M \twoheadrightarrow\beta\ M'$ then, to use the induction hypothesis on sn$M'$, we need to provide the requested hypotheses (1) through (4) correctly instantiated. (1) follows by Definition 2.17, and (2) and (4) are direct. As to (3), we need to prove that $\mathtt{Red}\,\beta\,(M'[P\,/\,x])$ holds for any Red$\alpha P$. By Lemma 3.3 we know that there exists some $R$ such that $M[P\,/\,x] \twoheadrightarrow\beta\ R$ and $R \sim\alpha\ M'[P\,/\,x]$. By hypothesis (3) it follows $\mathtt{Red}\,\beta\,(M[P\,/\,x])$, so by CR2 of Lemma 2.24 we obtain Red$\beta R$. Finally, we can use Lemma 3.3 together with inherited Lemma 2.21 to derive $\mathtt{Red}\,\beta\,(M'[P\,/\,x])$.
- Case appR: If $(\lambda xM)N \twoheadrightarrow\beta\ (\lambda xM)N'$ follows from $N \twoheadrightarrow\beta\ N'$ then, first, by Definition 2.17 we have sn$N'$, and secondly by CR2 of Lemma 2.24 we obtain Red$\alpha N'$, therefore we can use the induction hypothesis on sn$N'$ to derive $\mathtt{Red}\,\beta\,((\lambda xM)N')$.

□

So, to use the previous result in the main lemma, we are going to need a stronger induction hypothesis in order to derive the third hypothesis, namely $\forall\{P\} \to \mathtt{Red}\,\alpha\,P \to \mathtt{Red}\,\beta\,(M\,[\,P\,/\,x\,])$. We shall see that by stating the main lemma as next we can easily derive it:

**Lemma 3.5.** `main : ∀ {α M σ Γ} → Γ ⊢ M : α → RedSubst σ Γ → Red α (M • σ)`

*Proof.* By induction on the typing derivation:

- Case $\vdash$`var`: If $M$ is a variable, then the thesis follows directly from Definition 2.26.
- Case $\vdash$`abs`: If $M = \lambda xM'$ with type $\alpha \Rightarrow \beta$, then we need to show $\mathtt{Red}\,\beta\,(((\lambda xM') \bullet \sigma)N)$ for any $\mathtt{Red}\,\alpha\,N$. First of all, $\lambda xM' \bullet \sigma = \lambda y(M' \bullet (\sigma \prec\!+ (x\,,\,y)))$ for some fresh name $y$. Now, to use Lemma 3.4 we need to derive its hypothesis: (1) $\mathtt{sn}\,(M' \bullet \sigma \prec\!+ (x\,,\,y))$; (2) $\mathtt{sn}\,N$; (3) for every $\mathtt{Red}\,\alpha\,P$, $\mathtt{Red}\,\beta\,((M' \bullet \sigma \prec\!+ (x\,,\,y))\,[\,P\,/\,y\,])$, and; (4) $\mathtt{Red}\,\alpha\,N$. As to (1), by Lemma 2.28 we have $\mathtt{RedSubst}\,(\Gamma \Cup x : \alpha)\,(\sigma \prec\!+ (x\,,\,y))$, thus by induction hypothesis $\mathtt{Red}\,\beta\,(M' \bullet \sigma \prec\!+ (x\,,\,y))$, and so by `CR1` of Lemma 2.24 we obtain the desired result. (2) follows immediately by `CR1`. As to (3), first by Lemma 2.8 we have $(M' \bullet \sigma \prec\!+ (x\,,\,y))\,[\,P\,/\,y\,] \sim\alpha\ M' \bullet \sigma \prec\!+ (x\,,\,P)$. Next, by Lemma 2.28, $\mathtt{RedSubst}\,(\Gamma \Cup x : \alpha)\,(\sigma \prec\!+ (x\,,\,P))$, so by the induction hypothesis we have $\mathtt{Red}\,\beta\,(M'(\sigma, P/x))$. And finally, by Lemma 3.3 together with Lemma 2.21 we can derive the desired result. (4) is an assumption already made. At last, having (1) through (4) we can use Lemma 3.4 and derive $\mathtt{Red}\,(\alpha \Rightarrow \beta)\,((\lambda xM') \bullet \sigma)$, and so obtain $\mathtt{Red}\,\beta\,(((\lambda xM') \bullet \sigma)N)$ by Definition 2.20, as desired.
- Case $\vdash$`app`: Immediate by the induction hypothesis.

□

Without further ado, we have the Strong Normalization Theorem:

**Theorem 3.6.** `strongNormalization : ∀ {Γ M α} → Γ ⊢ M : α → sn M`

*Proof.* By Lemmas 2.27 and 3.5 we have $\mathtt{Red}\,\alpha\,(M \bullet \iota)$, and so by `CR1` of Lemma 2.24, $\mathtt{sn}\,(M \bullet \iota)$. Then, by Lemma 2.7, $M \bullet \iota \sim\alpha\ M$, and thus by Lemma 3.3 together with Lemma 2.19 it follows $\mathtt{sn}\,M$. □

# 4 System T

Let `C` and $\triangleright$`T` be inductively defined:

```
data C : Set where
  O : C; S : C; Rec : C

data _▷T_ : Rel where
  beta : ∀ {M N} → M ▷β N → M ▷T N
  recO : ∀ {G H} → k Rec · G · H · k O ▷T G
  recS : ∀ {G H N} → k Rec · G · H · (k S · N) ▷T H · N · (k Rec · G · H · N)
```

The syntax and theories of substitution, alpha- and beta-conversion for System T are then obtained by instantiating the framework with both `C` and $\triangleright$`T`, and similarly to STLC as shown in the previous section.

The assignment of types in System T is extended from STLC and defined:

```
data _⊢_:_ (Γ : Cxt) : Λ → Type → Set where
  ⊢zro : Γ ⊢ k O : nat
  ⊢suc : Γ ⊢ k S : nat ⇒ nat
  ⊢rec : ∀ {α} → Γ ⊢ k Rec : α ⇒ (nat ⇒ α ⇒ α) ⇒ nat ⇒ α
  ⊢var : ∀ {x} → (k : x ∈ Γ) → Γ ⊢ v x : Γ ⟨ k ⟩
  ⊢abs : ∀ {x M α β} → Γ ⋓ x : α ⊢ M : β → Γ ⊢ ƛ x M : α ⇒ β
  ⊢app : ∀ {M N α β} → Γ ⊢ M : α ⇒ β → Γ ⊢ N : α → Γ ⊢ M · N : β
```

`nat` is syntax-sugar for $\tau$.

## 4.1 The Strong Normalization Theorem in System T

The proof of the Strong Normalization Theorem in System T follows the same structure as the one of STLC: first, we need to find an appropriate definition of the neutral terms so to derive the first step in Girard's method, i.e., `CR1`, and then, we need to have a (main) lemma and reason by induction on the syntax (the typing judgment) to derive reducibiliy. Finally, the Strong Normalization Theorem in System T follows *exactly* as Theorem 3.6.

So, to start with, let the set of the neutral terms in System T be defined:

```
data NeT : Pred where
  var : ∀ {x} → NeT (v x)
  zro : NeT (k O)
  suc : NeT (k S)
  app : ∀ {M} → NeT M → ∀ {N} → NeT (M · N)
  beta : ∀ {x M N} → NeT (ƛ x M · N)
  neRec : ∀ {G H N} → NeT (k Rec · G · H · N)
```

One can easily convince himself that this definition also respects conditions in Definition 2.22. In addition to that, $\triangleright$`T` satisfies conditions in Definition 2.23, therefore we inherit Lemma 2.24 in System T, particularly `CR1`.

As to the second part, i.e., the main lemma, we have to consider only the additional syntax; the remaining cases follow identically. `O` and `S` are reducible by `CR3` (similar to $v_0$ in the proof of `CR1`). As to `Rec`, we shall follow the same strategy as in STLC and have a separate lemma, namely the recursion lemma. In the next section we cover this last case, while at the same time we present the announced simplification.

## 4.2 Recursion

In this section, first we introduce the principle of induction used in the proof for the recursion lemma presented in [9] and from our perspective, then we explain our simplification, and finally we formalize the proof.

We must prove that the neutral term `Rec`$GHN$ is reducible, for any reducible terms $G$, $H$ and $N$. First, we shall strengthen our induction hypothesis: by `CR1` we know that $G$, $H$ and $N$ are `sn`, so we can assume that these derivations are given as additional hypotheses. Also, we need some preparatory definitions: let $\nu(M)$, $\ell(M)$ and $\mathtt{nf}(M)$ be respectively the length of the longest reduction starting in $M$, the count of `S` symbols in $M$, and the normal form of the (strongly normalizing) term $M$. Now, to prove our thesis we shall proceed by induction on the *strict component-wise* order (henceforth, just

component-wise order) on the 4-tuple[7] $(\mathtt{sn}\,G, \mathtt{sn}\,H, \nu(N), \ell(\mathtt{nf}(N)))$, where in $\mathtt{sn}\,G$ and $\mathtt{sn}\,H$ we shall use the structural order of $\mathtt{sn}$, in $\nu(N)$ the complete order on natural numbers[8], and in $\ell(\mathtt{nf}(N))$ the structural order on natural numbers. As we did in Lemma 3.4, we are going to use `CR3` of Lemma 2.24 for the matter, and so we have to prove that every reductum of $\mathtt{Rec}\,G\,H\,N$ is reducible. There are five cases: (1) $\mathtt{Rec}\,G'\,H\,N$ with $G \twoheadrightarrow_\beta G'$, (2) $\mathtt{Rec}\,G\,H'\,N$ with $H \twoheadrightarrow_\beta H'$, (3) $\mathtt{Rec}\,G\,H\,N'$ with $N \twoheadrightarrow_\beta N'$, (4) $G$ with $N = \mathtt{0}$, and (5) $H\,N'\,(\mathtt{Rec}\,G\,H\,N')$ with $N = \mathtt{S}\,N'$. As to (1) and (2), we can directly use the induction hypothesis on $\mathtt{sn}\,G'$ and $\mathtt{sn}\,H'$. As to (3), we can suspect that $\nu(N') < \nu(N)$, and so we can proceed likewise. (4) is a hypothesis. As to (5), it is immediate that $\ell(\mathtt{nf}(N')) < \ell(\mathtt{nf}(\mathtt{S}\,N'))$.

We can simplify the induction schema used above by dispensing with $\mathtt{nf}$, and instead proceed by induction on the component-wise order of the 3-tuple $(\mathtt{sn}\,G, \mathtt{sn}\,H, (\nu(N), \ell(N)))$, where in $\mathtt{sn}\,G$ and $\mathtt{sn}\,H$ we use the same order as above, but in $(\nu(N), \ell(N))$ we use the *lexicographical order* on tuples[9]. As to cases (1), (2) and (4), the induction is the same. As to (3), we have already assumed that $\nu(N') < \nu(N)$, so we can use the (lexicographical-based) induction hypothesis on $(\nu(N'), \ell(N'))$, and disregard if $\ell(N')$ goes off. Finally, as to (5), on the one hand, it is immediate that $\ell(N') < \ell(\mathtt{S}\,N')$. On the other hand, we can also guess that $\nu(N') = \nu(\mathtt{S}\,N')$, therefore we can proceed by induction on $(\nu(\mathtt{S}\,N'), \ell(N'))$.

Now, to formalize the recursion lemma based on the last induction schema, first we need to give some definitions, as usual. Next is the function that computes the list of reductio for any given term $M$, while at the same time proves it is *sound*, i.e., every element of the list is actually a reductum of $M$. We present it in two separate parts, first `redAux`, which as the name suggest, is an auxiliary function, and then `reductio` which is the complete and desired operation (we omit some code):

```
1  redAux : (M : Term) → List (Σ[ N ∈ Term ](M →β N))
2  redAux (ƛ x M · N)                 = [ (M [ N / x ] , ...) ]
3  redAux (k Rec · G · H · k O)       = [ (G , ...) ]
4  redAux (k Rec · G · H · (k S · N)) = [ (H · N · (k Rec · G · H · N) , ...) ]
5  redAux _                           = []
6
7  reductio : (M : Term) → List (Σ[ N ∈ Term ](M →β N))
8  reductio (k _)   = []
9  reductio (v _)   = []
10 reductio (ƛ x M) = mapL (mapΣ (ƛ x) abs) (reductio M)
11 reductio (M · N) = redAux (M · N) ++ ... (reductio M) ++ ... (reductio N)
```

$\mathtt{mapΣ}$ is the function that given two other functions and a tuple, it applies each function to one of the components of the tuple. The purpose of the auxiliary function is to put together the cases of redexes, and apart from the `reductio` definition, so to have a cleaner treatment in the case of applications in the latter (see line 11).

The algorithm is also *complete*, i.e., it outputs all reductio of $M$, and its proof follows by induction on the derivation of any given reduction:

**Lemma 4.1.** `lemmaReductio : ∀ {M N} (r : M →β N) → (N , r) ∈' (reductio M)`

`∈'` is the standard relation of membership in lists.

We can use the list returned by `reductio` to develop an algorithm that computes our first ordinal $\nu$, i.e., the length of the longest reduction beginning in some strongly normalizing term $M$ given, by

[7] The component-wise order on a $n$-tuple is given by: $a_i <_i b \Rightarrow (a_0 \ldots, a_i \ldots, a_n) <_i (a_0 \ldots, b \ldots, a_n)$ for any $i$, $n$ and $b$.

[8] The complete order on natural numbers is the same as transitive closure of the structural order on them.

[9] The lexicographical order on a tuple is given by: $a < b \Rightarrow (a,c) < (b,d)$ and $b < c \Rightarrow (a,b) < (a,c)$ for any $a$, $b$, $c$, $d$.

recursively computing such a result for every reductum of $M$, then selecting the longest one, and finally adding one for the first step. Notice that the length of longest path and the height of the derivation tree of $\mathtt{sn}M$ are synonyms, so we shall use them interchangeably:

```
ν : ∀ {M} → sn M → ℕ
ν {M} (acc i) = 1 + max (mapL (λ{(N , M→N) → ν (i N M→N)}) (reductio M))
```

`max` is the function that returns the maximum element in a given *list*. The above definition is standard for computing the height of any inductive type, except for that `sn` has an infinitary premise. This means that we need to enumerate all possible applications to obtain every possible sub-tree. Since every term $M$ has a finite number of redexes, so there can only be finitely many applications of the premise, i.e., reductions $M \twoheadrightarrow_\beta N$ for some $N$, all of them being enumerated by the `reductio` algorithm, as proven in Lemma 4.1.

The height of $\mathtt{sn}N$ equals to the height of $\mathtt{sn}(\mathtt{S}N)$, as guessed at the start of this section. This is immediate since the prefix `S` does not add any redex to any reduction path:

**Lemma 4.2.** `lemmaSν : ∀ {M} (p : sn M) (q : sn (k S · M)) → ν p ≡ ν q`

*Proof.* By induction on either the derivation of $p$ or $q$. □

Next we have that the height of $\mathtt{sn}M$ decreases after a computation step is consumed, or in other words, every (immediate) sub-tree of $\mathtt{sn}M$ is strictly smaller. The name of the lemma is `lemmaStepν`, and its proof follows by properties of lists, and by using Lemma 4.1:

**Lemma 4.3.** `∀ {M N i} (p : sn M) → p ≡ acc i → (r : M →β N) → ν (i N r) < ν p`

Notice the *apparently* clumsy way it was stated. `i N r` is a proof of $\mathtt{sn}N$. To require such a proof as an argument would be inefficient since we already know $\mathtt{sn}M$ and $M \twoheadrightarrow_\beta N$. Instead, by asking for the argument `p ≡ acc i` we can obtain the infinitary premise `i` of $\mathtt{sn}M$ (this can be easily supplied afterwards with the constructor of $\equiv$, `refl`), and apply it to `N` and `r`, and so obtain the proof of $\mathtt{sn}N$.

Next is our second ordinal:

**Definition 4.4.** *$\ell$ `: Term → ℕ` is the function that counts the number of occurrences of the* `S` *symbol in any given term, and it is defined by recursion on the term.*

Finally, we have the recursion lemma. Let `<-lex` be the lexicographical order on tuples of $\mathbb{N}$. Then `Acc _<-lex_` is the type of pairs that are accessible by such an order. It is easy to prove that for any proof $p$ of $\mathtt{sn}N$ for some $N$, it follows $(\nu(p), \ell(N))$ is in the accessible part of the lexicographical order, hence such an argument can always be derived. Then:

```
lemmaRec : ∀ {α G H N} → sn G → sn H → (p : sn N) → Acc _<-lex_ (ν p , ℓ N)
         → Red α G → Red (nat ⇒ α ⇒ α) H → Red α (k Rec · G · H · N)
```

*Proof.* By induction on the derivations of $\mathtt{sn}G$ and $\mathtt{sn}H$, and on the lexicographical order of the tuple $(\nu(p), \ell(N))$[10]. As already said several times by now, we shall resort to `CR3` of Lemma 2.24 for the matter. So let us fast-forward til the reductum analysis:

- Case `rec0`: If $\mathtt{Rec}\,G\,H\,\mathtt{0} \twoheadrightarrow_\beta G$ then the result is a hypothesis.

[10] In Agda every function is structural recursive, and each one of them will successfully pass the type-checking phase if, put it simply, there exists a subset of the arguments such that for every recursive call in any of its definiens, at least one of the arguments is structurally smaller whilst the others remains the same. This is equivalent to saying that the induction is based on the component-wise order of any arrangement of such a subset, i.e., on a tuple made up of such arguments.

- Case recS: If $\mathtt{Rec}\,G\,H\,(\mathtt{S}\,N) \twoheadrightarrow_\beta H\,N\,(\mathtt{Rec}\,G\,H\,N)$ then we can apply the induction hypothesis, since we know both $\nu(N) = \nu(\mathtt{S}\,N)$ by Lemma 4.2, and $\ell(N) < \ell(\mathtt{S}\,N)$ immediately by definition of $\ell$, and so we obtain $\mathtt{Red}\,\alpha\,(\mathtt{Rec}\,G\,H\,N)$. Finally, by Definition 2.20 on $\mathtt{Red}\,(\mathtt{nat} \Rightarrow \alpha \Rightarrow \alpha)\,H$ we obtain $\mathtt{Red}\,\alpha\,(H\,N\,(\mathtt{Rec}\,G\,H\,N))$.
- Case appR: If $\mathtt{Rec}\,G\,H\,N \twoheadrightarrow_\beta \mathtt{Rec}\,G\,H\,N'$ follows from $N \twoheadrightarrow_\beta N'$, then by Lemma 4.3 we know that $\nu(N') < \nu(N)$, and so we can use the induction hypothesis to derive the desired result.
- Case appL: If the reduction follows from one either in $G$ or $H$, then we can proceed directly by the induction hypothesis.

□

# 5 Related work

In this development we have encoded the lambda terms using first-order abstract syntax (FOAS). In contrast, other approaches use *higher-order abstract syntax* (HOAS) [16], i.e., binders and variables are encoded using the same ones in the host language. These systems have the advantage that substitution is already defined. The first such mechanization of the theorem for STLC was presented in [8], and by using the ATS/LF logical framework [25]. However, the theory of (terminating) recursive functions using FOAS is more established, and there are plenty of programming languages that support them. This makes fairly easy to translate this mechanization to other system supporting standard principles of induction.

A second difference with existing work is that in this paper we have used named variables instead of *de-Bruijn indices* [4], e.g., in our framework the identity function can be written $\lambda xx$ for any $x$, while in the latter $\lambda 0$. Clearly, the former is visually more appealing, making it better suited for textbooks, needless to say it is the actual way programs are written. The main disadvantage is that we do not identify alpha-convertible terms, e.g., $\lambda v_0 v_0$ and $\lambda v_1 v_1$ are different objects, whereas by using indices there is only one possible representative for each class of alpha-convertible terms, and so it is not necessary to deal with alpha-conversion at all. To mention some renowned mechanizations of the theorem for STLC using this encoding: in [2] the author uses the LEGO system [17], and; in [1] two different mechanizations are presented, one in Agda and one in Coq [14].

As to System T, to the best of our knowledge there is not yet a formalization of the Strong Normalization Theorem.

# 6 Conclusions

We have presented a framework for the meta-theory of lambda calculi in FOAS with constants, that does not identify alpha-convertible terms, and it is parameterised by a reduction schema. On top of it, we have built a complete mechanization of Girard's proof of the Strong Normalization Theorem for System T. In addition, we were able to include a simplification on the principle of induction of the original proof. Finally, we gave a different mechanization of the same method but for STLC, and by using the new framework.

In terms of size, the framework is ∼1800LOC long, counting import statements and the like, and of which ∼90LOC belong to the first part of Girard's proof, namely the reducibility properties. As to the mechanizations of the proofs for STLC and System T, they are about 80 and 250LOC long repectively.

Shortly after [24], a refactorization of Joachimski and Matthes' syntactical mechanization for the Strong Normalization Theorem in STLC was made, resulting in a cleaner ∼400LOC. Clearly, the for-

malization presented in here is sharper; about a half if one takes the reducibility properties into consideration. One of the main differences is that the closure of the accessibility definition with alpha-conversion required just 3LOC, while its syntactical counterpart ∼100LOC[11]. Of course, this evaluation is subject to the emergence of an even cleaner version.

## References


[1] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer & Kathrin Stark (2019): *POPLMark reloaded: Mechanizing proofs by logical relations*. Journal of Functional Programming 29, doi:10.1017/S0956796819000170.

[2] Thorsten Altenkirch (1993): *Constructions, Inductive Types and Strong Normalization*. Ph.D. thesis, University of Edinburgh. Available at `https://www.cs.nott.ac.uk/~psztxa/publ/phd93.pdf`.

[3] Hendrik P. Barendregt (1985): *The lambda calculus - its syntax and semantics*. Studies in logic and the foundations of mathematics 103, North-Holland.

[4] Nicolaas Govert de Bruijn (1972): *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*. Indagationes Mathematicae 75(5), pp. 381–392, doi:10.1016/1385-7258(72)90034-0.

[5] Arthur Charguéraud (2012): *The Locally Nameless Representation*. Journal of Automated Reasoning 49(3), pp. 363–408, doi:10.1007/s10817-011-9225-2.

[6] Ernesto Copello, Nora Szasz & Álvaro Tasistro (2017): *Formal metatheory of the Lambda calculus using Stoughton's substitution*. Theoretical Computer Science 685, pp. 65–82, doi:10.1016/j.tcs.2016.08.025.

[7] Martín Copes, Nora Szasz & Álvaro Tasistro (2018): *Formalization in Constructive Type Theory of the Standardization Theorem for the Lambda Calculus using Multiple Substitution*. In Frédéric Blanqui & Giselle Reis, editors: *Proc. LFMTP '18*, *EPTCS* 274, Open Publishing Association, p. 27–41, doi:10.4204/eptcs.274.3.

[8] Kevin Donnelly & Hongwei Xi (2007): *A Formalization of Strong Normalization for Simply-Typed Lambda-Calculus and System F*. In Alberto Momigliano & Brigitte Pientka, editors: *Proc. LFMTP '06*, *ENTCS* 174, Elsevier, pp. 109–125, doi:10.1016/j.entcs.2007.01.021.

[9] Jean-Yves Girard, Paul Taylor & Yves Lafont (1989): *Proofs and Types*. Cambridge University Press.

[10] V. Kurt Gödel (1958): *Über Eine Bisher Noch Nicht Benützte Erweiterung des Finiten Standpunktes*. Dialectica 12(4), pp. 280–287, doi:10.1111/j.1746-8361.1958.tb01464.x.

[11] J. Roger Hindley & Jonathan P. Seldin (2008): *Lambda-Calculus and Combinators: An Introduction*, second edition. Cambridge University Press.

[12] Felix Joachimski & Ralph Matthes (2003): *Short proofs of normalization for the simply-typed λ-calculus, permutative conversions and Gödel's T*. Archive for Mathematical Logic 42, pp. 59–87, doi:10.1007/s00153-002-0156-9.

[13] Per Martin-Löf & Giovanni Sambin (1984): *Intuitionistic type theory*. Studies in proof theory, Bibliopolis.

[14] The Coq development team (2004): *The Coq proof assistant reference manual*. Available at `http://coq.inria.fr`.

[15] Ulf Norell (2007): *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology.

[16] Frank Pfenning & Conal Elliott (1988): *Higher-Order Abstract Syntax*. ACM SIGPLAN Notices 23(7), p. 199–208, doi:10.1145/53990.54010.

[17] Robert Pollack (1994): *The Theory of LEGO*. Ph.D. thesis, University of Edinburgh.


[11] These figures do not include neither the commutativity nor the compatibility properties in any of both parties.

[18] Allen Stoughton (1988): *Substitution revisited*. Theoretical Computer Science 59(3), pp. 317–325, doi:10.1016/0304-3975(88)90149-1.

[19] Aaron Stump (2016): *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, doi:10.1145/2841316.

[20] William W. Tait (1967): *Intensional Interpretations of Functionals of Finite Type I*. The Journal of Symbolic Logic 32(2), pp. 198–212, doi:10.2307/2271658.

[21] Álvaro Tasistro, Ernesto Copello & Nora Szasz (2015): *Formalisation in Constructive Type Theory of Stoughton's Substitution for the Lambda Calculus*. In Mauricio Ayala-Rincón & Ian Mackie, editors: *Proc. LSFA '14*, *ENTCS* 312, Elsevier, pp. 215–230, doi:10.1016/j.entcs.2015.04.013.

[22] The Agda development team (2011): *The Agda standard library*. Available at `https://github.com/agda/agda-stdlib`.

[23] Christian Urban, Andrew M. Pitts & Murdoch J. Gabbay (2004): *Nominal unification*. Theoretical Computer Science 323(1), pp. 473–497, doi:10.1016/j.tcs.2004.06.016.

[24] Sebastián Urciuoli, Álvaro Tasistro & Nora Szasz (2020): *Strong Normalization for the Simply-Typed Lambda Calculus in Constructive Type Theory Using Agda*. In Cláudia Nalon & Giselle Reis, editors: *Proc. LSFA '20*, *ENTCS* 351, Elsevier, pp. 187–203, doi:10.1016/j.entcs.2020.08.010.

[25] Hongwei Xi (2003): *Applied type system*. In Stefano Berardi, Mario Coppo & Ferruccio Damiani, editors: *Proc. TYPES '03*, *LNCS* 3085, Springer, pp. 394–408, doi:10.1007/978-3-540-24849-1_25.