Tenth Workshop on Logical and Semantic Frameworks, with Applications

LSFA 2015

Natal, Brazil

August 31 – September 1, 2015

Preliminary Proceedings

Edited by Mario Benevides and René Thiemann
Selected and revised papers will be published in
Electronic Notes in Theoretical Computer Science.
Contents

Preface ......................................................... v

Valeria de Paiva (Invited Speaker)
   Modal Type Theory ........................................ 1

Andrés Viso, Eduardo Bonelli, and Mauricio Ayala-Rincón
   Type Soundness for Path Polymorphism .................... 3

Mauricio Ayala-Rincón, Maribel Fernández, and Ana Cristina Rocha-Oliveira
   Completeness in PVS of a Nominal Unification Algorithm .... 19

Antonio Bucciarelli, Delia Kesner, and Daniel Ventura
   Strong Normalization through Intersection Types and Memory ...... 35

Ernesto Copello, Álvaro Tasistro, Nora Szasz, Ana Bove, and Maribel Fernández
   Alpha-Structural Induction and Recursion for the Lambda Calculus in Constructive Type Theory ........................................ 51

Dale Miller (Invited Speaker)
   Defining the semantics of proof evidence ....................... 67

Elaine Pimentel, Vivek Nigam, and João Neto
   Multi-focused Proofs with Different Polarity Assignments .......... 69

Marcela Quispe-Cruz, Edward Hermann Haeusler, and Lew Gordeev
   On Strong Normalization in Proof-graphs for Propositional Logic ...... 85

Laís Andrade, Ruan Carvalho, Anjolina de Oliveira, and Ruy J. G. B. de Queiroz
   Normalization of N-Graphs via Sub-N-Graphs ..................... 101

Paulo A. S. Veloso and Sheila R. M. Veloso
   On Graphs for Intuitionistic Modal Logics ...................... 117

Valentin Goranko (Invited Speaker)
   A logical framework for multi-agent visual-epistemic reasoning ........ 133

João Marcos and Cláudia Nalon
   Classical resolution for many-valued logics ..................... 135
Arthur F. Ramos, Ruy J. G. B. de Queiroz, and Anjolina G. de Oliveira
Computational paths, identity type, and the groupoid model ........... 151

Marcus V. M. Ramos and Ruy J. G. B. de Queiroz
Formalization of simplification for context-free grammars ............ 167

Benjamín Callejas Bedregal
Nondeterministic Linear Automata and a Class of Deterministic Linear Languages .......................................................... 183

Ofer Arieli (Invited Speaker)
Sequent-Based Argumentation .................................................... 197

Mauricio Ayala-Rincón, Maribel Fernández, Murdoch James Gabbay, and Ana Cristina Rocha-Oliveira
Checking Overlaps of Nominal Rewriting Rules .................... 199

Leonardo Rodríguez, Miguel Pagano, and Daniel Fridlender
Proving Correctness of a Compiler Using Step-indexed Logical Relations 215

Roy L. Crole and Amy Furniss
Canonical HybridLF: Extending Hybrid with Dependent Types ...... 231
Preface

This volume contains the preliminary proceedings of the Tenth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2015). The workshop was held in Natal, Brazil on August 31 – September 1, 2015 as part of NAT@Logic 2015.


The aim of LSFA is to bring together researchers and students interested in theoretical and practical aspects of logical and semantic frameworks and their applications. This included topics such as proof, type and rewriting theory, specification and deduction languages, formal semantics of languages and systems, among others.

In total there have been 19 submissions of which 16 have been completed. From these 16 submissions, 13 were accepted as regular presentations. Moreover, 2 submissions have been invited as a single shared presentation, and 1 submission was accepted as a short presentation. Each submission was reviewed by at least three reviewers, and an electronic Program Committee (PC) meeting was held using the EasyChair system. The reviews were written by the 30 PC members and 16 additional reviewers. We thank the PC members and the additional reviewers for doing a great job in writing high-quality reviews and active discussions.

All submissions which have been presented at LSFA will be invited to submit a revised version of their paper for the ENTCS post-proceedings, taking into account the previous reviews and the comments during the workshop. To this end, there will be another round of refereeing, which will be more selective than the one for presentation.

Besides the regular submissions, LSFA 2015 includes four invited talks by Ofer Arieli (Sequent-Based Argumentation), by Valentin Goranko (A logical framework for multi-agent visual-epistemic reasoning), by Dale Miller (Defining the semantics of proof evidence), and by Valeria de Paiva (Modal Type Theory). Abstracts of each the invited speakers are contained in these preliminary proceedings.

We would like to thank all those who contributed to LSFA 2015, and in particular the invited speakers for kindly accepting to present their work at LSFA 2015. We are very grateful to the local organizers, especially to Elaine Pimentel and João Marcos, for taking care of all the local organization and for their efficient support. Finally, we appreciate the financial support provided by CNPq, CAPES, DMAT/UFRN, PPgMAE/UFRN, PPgSC/UFRN, and the EU FP7 Marie Curie PIRSES-GA-2012-318986 project GeTFun.

Mario Benevides and René Thiemann

August 2015
Programme Committee

- Wolfgang Ahrendt (Chalmers University of Technology, Sweden)
- Takahito Aoto (Tohoku University, Japan)
- Mauricio Ayala-Rincon (Universidade de Brasília, Brazil)
- Verónica Becher (Universidad de Buenos Aires, Argentina)
- Mario Benevides (Federal University of Rio de Janeiro, Brazil, co-chair)
- Jasmin Blanchette (Inria Nancy, France)
- Walter Carnielli (State University of Campinas, Brazil)
- Agata Ciabattoni (Vienna University of Technology, Austria)
- Flávio L. C. de Moura (Universidade de Brasília, Brazil)
- Ruy de Queiroz (Universidade Federal de Pernambuco, Brazil)
- Catherine Dubios (ENSIIE, France)
- Marcelo Finger (University of Sao Paulo, Brazil)
- Mario Florido (University of Porto, Portugal)
- Jürgen Giesl (RWTH Aachen University, Germany)
- Edward Hermann Haeusler (Pontificia Universidade Catolica do Rio de Janeiro, Brazil)
- Andreas Herzig (Université Paul Sabatier, France)
- Delia Kesner (Université Paris Diderot, France)
- Ian Mackie (École Polytechnique, France)
- Cesar Muñoz (NASA Langley Research Center, USA)
- Vivek Nigam (Universidade Federal da Paraíba, Brazil)
- Elaine Pimentel (DMAT, Universidade Federal do Rio Grande do Norte, Brazil)
- Camilo Rocha (Escuela Colombiana de Ingeniería, Colombia)
- Manfred Schmidt-Schauß (Goethe University, Germany)
- Alvaro Tasistro (Universidad ORT, Uruguay)
- René Thiemann (University of Innsbruck, Austria, co-chair)
- Hans van Ditmarsch (LORIA Nancy, France)
- Femke van Raamsdonk (VU Amsterdam, Netherlands)
- Ivan Varzinczak (Universidade Federal do Rio de Janeiro, Brazil)
- Sheila Veloso (Universidade Federal do Rio de Janeiro, Brazil)
- Freek Wiedijk (Radboud University, Netherlands)
Modal Type Theory

Valeria de Paiva

Nuance Communications, USA

Abstract
Type theory does not usually discuss logical modalities, and modalities tend to be mostly studied within classical logic, not type theory. But modalities should be useful in type theory, as they are generally very useful in theoretical computer science. Since there seems to be a renewed interest in the notions of constructive modal type theory and linear type theory, in part caused by the interest in homotopy type theory, it seems sensible to recapitulate some of the known facts, especially ones on the semantics of modal type theory. I will describe a fibrational categorical semantics for the necessity-only fragment of constructive modal type theory, both with and without dependent types. Dependent type theories are usually, but not always, given categorical semantics in terms of fibrations. We provide semantics in terms of fibrations for both the non-dependent and the dependent modal type systems proposed and prove them sound and complete. We also discuss some of the possible alternatives. (This is joint work with Eike Ritter, Birmingham).

Keywords: Modal logics, graph calculus, Kripke semantics, special modalities, refutation.

1 Email: valeria.depaiva@gmail.com

Selected, revised papers will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
Type Soundness for Path Polymorphism

Andrés Viso\textsuperscript{1}

CONICET and Universidad de Buenos Aires (Argentina)

Eduardo Bonelli\textsuperscript{2}

CONICET and Universidad Nacional de Quilmes (Argentina)

Mauricio Ayala-Rincón\textsuperscript{3}

Universidade de Brasília (Brazil)

Abstract

Path polymorphism is the ability to define functions that can operate uniformly over arbitrary recursively
specified data structures. Its essence is captured by patterns of the form \(xy\) which decompose a compound
into its parts. Typing these kinds of patterns is challenging since, from the classical approach, the type of
the compound does not determine the type of its components. We propose a static type system (i.e. no run-
time analysis) for a pattern calculus that captures this feature. Our solution combines type application,
constants as types, union types and recursive types. We address the fundamental properties of subject
reduction and progress that guarantee a well-behaved dynamics. Both these results rely crucially on a
notion of pattern compatibility and also on a coinductive characterisation of subtyping.

Keywords: \(\lambda\)-Calculus, Pattern Matching, Path Polymorphism, Static Typing

1 Introduction

Applicative representation of data structures in functional programming languages
consists in applying variable arity constructors to arguments. Examples are:

\[
s = \text{cons}(v_1 v_1) (\text{cons}(v_1 v_2) \text{nil})
\]

\[
t = \text{node}(v_1 v_3) (\text{node}(v_1 v_4) \text{nil}\text{nil}) (\text{node}(v_1 v_5) \text{nil}\text{nil})
\]

These are data structures that hold values, prefixed by the constructor \texttt{vl} for “value”
\((v_{1,2} \text{ in the first case, and } v_{3,4,5} \text{ in the second}). Consider the following function for

\text{Selected, revised papers will be published in}\nElectronic Notes in Theoretical Computer Science
URL: \texttt{www.elsevier.nl/locate/entcs}
Viso, Bonelli, and Ayala-Rincon

Updating the values of any of these two structures by applying some user-supplied function \( f \) to it:

\[
\text{ upd } = f \rightarrow_{\{ f: A \supset B \}} (v1 z \rightarrow_{\{ z:A \}} v1 (f z)) \\
| xy \rightarrow_{\{ x:C; y:D \}} (\text{ upd } f x) (\text{ upd } f y) \\
| w \rightarrow_{\{ w:E \}} w)
\]

Both \( \text{ upd } (+1) s \) and \( \text{ upd } (+1) t \) may be evaluated. The expression to the right of “\( = \)” is called an abstraction and consists of a unique branch; this branch in turn is formed from a pattern \( (f) \), a user-specified type declaration for the variables in the pattern \( \{ f: A \supset B \} \), and a body (in this case the body is itself another abstraction that consists of three branches). An argument to an abstraction is matched against the patterns, in the order in which they are written, and the appropriate body is selected. Notice the pattern \( xy \). This pattern embodies the essence of what is known as path polymorphism \cite{15,17} since it abstracts a path being “split”. The starting point of this paper is how to type a calculus, let us call it \( \text{ CAP } \) for Calculus of Applicative Patterns, that admits such examples. We next show why the problem is challenging, explain our contribution and also discuss why the current literature falls short of addressing it. We do so with an introduction-by-example approach, for the full syntax and semantics of the calculus refer to Sec. 2.

**Preliminaries on typing patterns expressing path polymorphism**

Consider these two simple examples:

\[
(c \rightarrow 1) d \\
(c x \rightarrow_{\{ x:\text{Nat} \}} x + 1) (c \text{true})
\]

They should clearly not be typable. In the first case, the abstraction is not capable of handling \( d \). This is avoided by introducing singleton types in the form of the constructors themselves: \( c \) is given type \( c \) while \( d \) is given type \( d \); these are then compared. In the second case, \( x \) in the pattern is required to be \( \text{Nat} \) yet the type of the argument to \( c \) in \( c \text{true} \) is \( \text{Bool} \). This is avoided by introducing type application \cite{22} into types: \( c x \) is assigned a type of the form \( c \at \text{Nat} \) while \( c \text{true} \) is assigned type \( c \at \text{Bool} \); these are then compared.

Consider next the pattern \( xy \) of \( \text{ upd } \). It can be instantiated with different applicative terms in each recursive call to \( \text{ upd } \). For example, suppose \( A = B = \text{Nat} \), that \( v_1 \) and \( v_2 \) are numbers and consider \( \text{ upd } (+1) s \). The following table illustrates some of the terms with which \( x \) and \( y \) are instantiated during the evaluation of \( \text{ upd } (+1) s \):

<table>
<thead>
<tr>
<th>( \text{ upd } (+1) s )</th>
<th>( x )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ cons } (v1 v1) )</td>
<td>( \text{ cons } (v1 v2) )</td>
<td>( \text{ nil } )</td>
</tr>
<tr>
<td>( \text{ cons } (v1 v1) )</td>
<td>( \text{ cons } (v1 v2) )</td>
<td>( \text{ nil } )</td>
</tr>
<tr>
<td>( \text{ cons } (v1 v2) )</td>
<td>( \text{ cons } (v1 v1) )</td>
<td>( \text{ nil } )</td>
</tr>
</tbody>
</table>
The type assigned to $x$ (and $y$) should encompass all terms in its respective column. This suggests adopting a union type for $x$. On the assumption that the programmer has provided an exhaustive coverage, the type of $x$ in $\text{upd}$ is:

$$\mu \alpha. (\text{vl} @ A) \oplus (\alpha @ \alpha) \oplus (\text{cons} \oplus \text{node} \oplus \text{nil})$$

Here $\mu$ is the recursive type constructor and $\oplus$ the union type constructor. The variable $y$ in the pattern $xy$ will also be assigned the same type.

Here $\text{upd}$ itself is assigned type $(A \supset B) \supset (F_A \supset F_B)$, where $F_X$ is $\mu \alpha. (\text{vl} @ X) \oplus (\alpha @ \alpha) \oplus (\text{cons} \oplus \text{node} \oplus \text{nil})$. Thus variables in applicative patterns will be assigned union types.

Recursive types are useful to give static semantics to fixpoints and, together with unions, they allow to model recursively defined data types. Combining this idea with type application allows to define data types in a more intuitive manner, like for example lists and trees

$$\mu \alpha. \text{nil} \oplus (\text{cons} @ A @ \alpha)$$

The advantage of this approach is that the type expression reflects the structure of the terms that inhabit it (cf. Fig. 3). This will prove to be very convenient for our proposed notion of pattern compatibility.

Compatibility is the key for ensuring Safety (Subject Reduction, SR for short, and Progress). Consider the following example:

$$\begin{aligned}
(c \; x \to_{\{x:\text{Bool}\}} \; \text{if} \; x \; \text{then} \; 1 \; \text{else} \; 0) \mid (c \; y \to_{\{y:\text{Nat}\}} \; y + 1)
\end{aligned}$$

Although there is a branch capable of handling a term such as $c\,4$, namely the second one, evaluation in CAP takes place in left-to-right order following standard practice in functional programming languages. Since the term $c\,4$ also matches the pattern $c\,x$, we would obtain the (incorrect) reduct $\text{if} \; 4 \; \text{then} \; 1 \; \text{else} \; 0$. We thus must relate the types of $c\,x$ and $c\,y$ in order to avoid failure of SR. Since $c\,y$ is an instance of $c\,x$, we require the type of the latter to be a subtype of the type of the former since it will always have priority: $c @ \text{Nat} \preceq c @ \text{Bool}$. Fortunately, this is not the case since $\text{Nat} \not\preceq \text{Bool}$, rendering this example untypable.

Consider now, a term such as:

$$f \to_{\{f:\text{A} \supset \text{B}\}} (\; c \; z \to_{\{z:\text{A}\}} c \; (f \; z) \; \mid \; x \; y \to_{\{x:\text{C}, y:\text{D}\}} x \; y)$$

This function takes an argument $f$ and pattern-matches with a data structure to apply $f$ only when this data structure is an application with the constructor $c$ on the left-hand side. Assigning $x$ in the second branch the type $C = c$ is a potential source of failure of SR since the function would accept arguments of type $c @ D$. Our proposed notion of compatibility will check the types occurring at offending positions in the types of both patterns. In this case, if $C = c$ then $C @ D \preceq c @ A$ is enforced. Note that if $C$ were a type such as $\mu \alpha. c \oplus d @ \alpha$, then also the same condition would be enforced.
Let us return to example (1). The type declarations would be \( C = D = \mu \alpha . (v \uplus A) \ominus (\alpha @ a) \ominus (\text{cons} \ominus \text{node} \ominus \text{nil}) \) and \( E = \text{cons} \ominus \text{node} \ominus \text{nil} \). We now illustrate how compatibility determines any possible source of failure of SR. Let us call \( p, q \) and \( r \) the three patterns of the innermost abstraction of (1), resp. Since pattern \( p \) does not subsume \( q \), we determine the (maximal) positions in both patterns which are sources of failure of subsumption. In this case, it is that of \( v \uplus A \) in \( p \) and \( x \) in \( q \). We now consider the subtype at that position in \( v \uplus A \), the type of \( p \), and the subtype at the same position in \( F_A \uplus F_A \), the type of \( q \): the first is \( v \uplus A \) and the second is \( F_A \). Since \( F_A \) does not admit \( v \uplus A \) (cf. Def. 3.5), these branches are immediately declared compatible. In the case of \( p \) and \( r \), \( \epsilon \) is the offending position in the failure of \( p \) subsuming \( r \): since the type application constructor \( @ \) located at position \( \epsilon \) in \( v \uplus A \) is not admitted by \( E \), the type of \( r \), these branches are immediately declared compatible. Finally, a similar analysis between \( q \) and \( r \) entails that these are compatible too. The type system and its proof of Safety will therefore assure us that this example preserves typability.

**Summary of contributions:**

- A typing discipline for CAP. We statically guarantee safety for path polymorphism in its purest form (other, more standard forms of polymorphism such as parametric polymorphism which we believe to be easier to handle, are out of the scope of this paper).

- A proof of safety for the resulting system. It relies on the syntactic notion of pattern compatibility mentioned above, hence no runtime analysis is required.

- Invertibility of subtyping of recursive types. This is crucial for the proof of safety. It relies on an equivalent coinductive formulation for which invertibility implies invertibility of subtyping of recursive types.

**Related work**

The literature on (typed) pattern calculi is extensive; we mention the most relevant ones (see [15, 17] for a more thorough listing). In [2] the constructor calculus is proposed. It has a different notion of pattern matching: it uses a case construct \( \{c_1 \mapsto s_1, \ldots, c_n \mapsto s_n\} \cdot t \) in which certain occurrences of the constructors \( c_i \) in \( t \) are replaced by their corresponding terms. [22] studies typing to ensure that these constructor substitutions never block on a constant not in their domain. Recursive types are not considered (nor is path polymorphism). Two further closely related efforts merit comments: the first is the work by Jay and Kesner and the second is that of the \( \rho \)-calculus by Kirchner and colleagues.

In [16, 17] the Pure Pattern Calculus (PPC) is studied. It allows patterns to be computed dynamically (they may contain free variables). A type system for a PPC like calculus is given in [15] however neither recursive nor union types are considered. [15] also studies a simple static pattern calculus. However, there are numerous differing aspects w.r.t. this work among which we can mention. First, the typed version of [15] (the Query Calculus) omits recursive types and union types. Then, although it admits a form of path polymorphism, this is at the cost of matching types at runtime and thus changing the operational semantics of the
untyped calculus; our system is purely static, no runtime analysis is required.

The \( \rho \)-calculus [9] is a generic pattern matching calculus parameterized over a matching theory. There has been extensive work exploring numerous extensions [5, 10–13, 20]. None addresses path polymorphism however. Indeed, none of the above allow patterns of the form \( xy \). This limitation seems to be due to the alternative approach to typing \( c x \) adopted in the literature on the \( \rho \)-calculus where \( c \) is assigned a fixed functional type. This approach seems incompatible with path polymorphism, as we see it, in that it suggests no obvious way of typing patterns of the form \( xy \) where \( x \) denotes an arbitrary piece of unstructured data. Additional differences with our work are:

- [12]: It does not introduce union types. No runtime matching error detection takes place (this is achieved via Progress in our paper).
- [10]: It deals with an untyped \( \rho \)-calculus. Hence no SR.
- [5, 11]: Neither union nor recursive types are considered.

Structure of the paper. Sec. 2 introduces the terms and operational semantics of CAP. The typing system is developed in Sec. 3 together with a precise definition of compatibility. Sec. 4 studies Safety: SR and Progress. For the benefit of the reviewing process a full report with all details of the proofs is available online [25].

2 Syntax and Operational Semantics of CAP

We assume given an infinite set of term variables \( \mathcal{V} \) and constants \( \mathcal{C} \). The syntax of CAP consists of four syntactic categories, namely patterns \( (p, q, \ldots) \), terms \( (s, t, \ldots) \), data structures \( (d, e, \ldots) \) and matchable forms \( (m, n, \ldots) \):

\[
\begin{align*}
\text{Patterns:} & \quad p ::= x \quad \text{(matchable)} \quad t ::= x \quad \text{(variable)} \\
& \quad | \quad c \quad \text{(constant)} \quad | \quad c \quad \text{(constant)} \\
& \quad | \quad pp \quad \text{(compound)} \quad | \quad tt \quad \text{(application)} \\
& \quad \quad | \quad p \to_{\theta} t \quad | \ldots \quad | \quad p \to_{\theta} t \quad \text{(abstraction)} \\
\text{Data Structures:} & \quad d ::= c \quad \text{(constant)} \quad m ::= d \quad \text{(data structure)} \\
& \quad | \quad dt \quad \text{(compound)} \quad | \quad p \to_{\theta} t \quad | \ldots \quad | \quad p \to_{\theta} t \quad \text{(abstraction)}
\end{align*}
\]

The set of patterns, terms, data structures and matchable forms are denoted \( \mathcal{P}, \mathcal{T}, \mathcal{D} \) and \( \mathcal{M} \), resp. Variables occurring in patterns are called matchables. We often abbreviate \( p_1 \to_{\theta_1} s_1 \ | \ldots \ | \ p_n \to_{\theta_n} s_n \) with \( (p_i \to_{\theta_i} s_i)_{i \in 1 \ldots n} \). The \( \theta_i \) are typing contexts annotating the type assignments for the variables in \( p_i \) (cf. Sec. 3). The free variables of a term \( t \) (notation \( \mathsf{fv}(t) \)) are defined as expected; in a pattern \( p \) we call them free matchables (\( \mathsf{fm}(p) \)). All free matchables in each \( p_i \) are assumed to be bound in their respective bodies \( s_i \). Positions in patterns and terms are defined as expected and denoted \( \pi, \pi', \ldots \) (\( \epsilon \) denotes the root position). We write \( \mathsf{pos}(s) \) for the set of positions of \( s \) and \( s|_{\pi} \) for the subterm of \( s \) occurring at position \( \pi \).

A substitution \((\sigma, \sigma_{i}, \ldots)\) is a partial function from term variables to terms. If
it assigns \( u_i \) to \( x_i, i \in 1..n \), then we write \( \{ u_1/x_1, \ldots, u_n/x_n \} \). Its domain \( \text{dom}(\sigma) \) is \( \{ x_1, \ldots, x_n \} \). Also, \( \{ \} \) is the identity substitution. We write \( \sigma s \) for the result of applying \( \sigma \) to term \( s \). 

Matchable forms are required for defining the matching operation, described next.

Given a pattern \( p \) and a term \( s \), the matching operation \( \{ s/p \} \) determines whether \( s \) matches \( p \). It may have one of three outcomes: success, fail (in which case it returns the special symbol \( \text{fail} \)) or undetermined (in which case it returns the special symbol \( \text{wait} \)). We say \( \{ s/p \} \) is decided if it is either successful or it fails. In the former it yields a substitution \( \sigma \); in this case we write \( \{ s/p \} = \sigma \).

The disjoint union of matching outcomes is given as follows ("\( \triangleq \)" is used for definitional equality):

\[
\text{fail} \cup o \triangleq \text{fail} \quad \text{wait} \cup \sigma \triangleq \text{wait} \\
o \cup \text{fail} \triangleq \text{fail} \quad \sigma \cup \text{wait} \triangleq \text{wait} \\
\sigma_1 \cup \sigma_2 \triangleq \sigma \quad \text{wait} \cup \text{wait} \triangleq \text{wait}
\]

where \( o \) denotes any possible output and \( \sigma_1 \cup \sigma_2 \triangleq \sigma \) if the domains of \( \sigma_1 \) and \( \sigma_2 \) are disjoint. To ensure this always holds we assumed patterns to be linear (at most one occurrence of any matchable). The matching operation is defined as follows, where the defining clauses below are evaluated from top to bottom 4:

\[
\begin{align*}
\{ u/x \} & \triangleq \{ u/x \} \\
\{ c/c \} & \triangleq \{ \} \\
\{ uv/pq \} & \triangleq \{ u/p \} \cup \{ v/q \} \quad \text{if} \ uv \text{ is a matchable form} \\
\{ u/p \} & \triangleq \text{fail} \quad \text{if} \ u \text{ is a matchable form} \\
\{ u/p \} & \triangleq \text{wait} \\
\end{align*}
\]

For example: \( \{ x \rightarrow s/c \} = \text{fail} \); \( \{ d/c \} = \text{fail} \); \( \{ x/c \} = \text{wait} \) and \( \{ c/c/x \ d \} = \text{fail} \). We now turn to the only reduction axiom of CAP:

\[
\begin{align*}
\{ u/p_i \} = \text{fail} \text{ for all } i < j \\
\{ u/p_j \} = \sigma_j \quad j \in 1..n \\
(p_i \rightarrow \theta_i \ s_i)_{i \in 1..n} u \rightarrow \sigma_j s_j
\end{align*}
\]

It may be applied under any context and states that if the argument \( u \) to an abstraction \( (p_i \rightarrow \theta_i \ s_i)_{i \in 1..n} \) fails to match all patterns \( p_i \) with \( i < j \) and successfully matches pattern \( p_j \) (producing a substitution \( \sigma_j \)), then the term \( (p_i \rightarrow \theta_i \ s_i)_{i \in 1..n} u \) reduces to \( \sigma_j s_j \).

The following example illustrates the use of the reduction rule and the matching

---

4 This is simplification to the static patterns case of the matching operation introduced in [17].
operation:
\[
(t_{\text{true}} \rightarrow 1 \mid t_{\text{false}} \rightarrow 0) ((t_{\text{true}} \rightarrow \text{false}) \mid (t_{\text{false}} \rightarrow \text{true}) \rightarrow \text{true}) \\
\rightarrow (t_{\text{true}} \rightarrow 1 \mid t_{\text{false}} \rightarrow 0) \{t_{\text{true}}/t_{\text{true}}\} \rightarrow \text{false} \\
= (t_{\text{true}} \rightarrow 1 \mid t_{\text{false}} \rightarrow 0) \rightarrow \{\text{false}/\text{false}\} \rightarrow 0 \\
\{\text{false}/\text{true}\} = \text{fail} \\
\rightarrow \{\text{false}/\text{false}\} \rightarrow 0 \\
\{\text{false}/\text{true}\} = 0
\]

Proposition 2.1 Reduction in CAP is confluent (CR).

This result follows from a straightforward adaptation of the CR proof presented in [17] to our calculus. The key step is proving that the matching operation satisfies the Rigid Matching Condition (RMC) proposed in the cited work. Note that CAP is just the static patterns fragment of PPC where instead of the usual abstraction we have alternatives (i.e. we abstract multiple branches with the same constructor). Our contribution is on the typed variant of the calculus.

3 Typing System

This section presents $\mu$-types, the finite type expressions that shall be used for typing terms in CAP, their associated notions of equivalence and subtyping and then the typing schemes. Also, further examples and definitions associated to compatibility are included.

3.1 Types

In order to ensure that patterns such as $xy$ decompose only data structures rather than arbitrary terms, we shall introduce two sorts of typing expressions: types and datatypes, the latter being strictly included in the former.

We assume given countably infinite sets $V_D$ of datatype variables ($\alpha, \beta, \ldots$), $V_A$ of type variables ($X, Y, \ldots$) and $C$ of type constants ($\varepsilon, d, \ldots$). We define $\mathcal{V} \equiv V_A \cup V_D$ and use metavariables $V, W, \ldots$ to denote an arbitrary element in it. Likewise, we write $a, b, \ldots$ for elements in $\mathcal{V} \cup C$. The sets $T_D$ of $\mu$-datatypes and $T$ of $\mu$-types, resp., are inductively defined as follows:

\[
\begin{align*}
D &::= \alpha & (\text{datatype variable}) & A &::= X & (\text{type variable}) \\
& \mid \varepsilon & (\text{atom}) & & D & (\text{datatype}) \\
& \mid D @ A & (\text{compound}) & A &\supset A & (\text{type abstraction}) \\
& \mid D \oplus D & (\text{union}) & A &\oplus A & (\text{union}) \\
& \mid \mu\alpha.D & (\text{recursion}) & & \mu X.A & (\text{recursion})
\end{align*}
\]

Remark 3.1 A type of the form $\mu\alpha.A$ is not valid in general since it may produce invalid unfoldings. For example, $\mu\alpha.\alpha \supset \alpha = (\mu\alpha.\alpha \supset \alpha) \supset (\mu\alpha.\alpha \supset \alpha)$, since $\alpha$ is a datatype variable and type abstraction is not a datatype. On the other hand, types of the form $\mu X.D$ are not necessary since they denote the solution to the equation $X = D$, hence $X$ is a variable representing a datatype.
(E-IDEM) \[ \vdash A \oplus A \simeq_\mu A \]
(E-COMM) \[ \vdash A \oplus B \simeq_\mu B \oplus A \]
(E-FOLD) \[ \vdash \mu V.A \simeq_\mu \{ \mu V.A/V \} A \]
(E-ASSOC) \[ \vdash A \oplus (B \oplus C) \simeq_\mu (A \oplus B) \oplus C \]
(E-CONTR) \[ \vdash A \simeq_\mu \{ A/V \} B \quad \mu V.B \text{ contractive} \]

Fig. 1. Type equivalence for \( \mu \)-types (sample)

We consider \( \oplus \) to bind tighter than \( \supset \), while \( \oplus \) binds tighter than \( \oplus \). Therefore \( D \oplus A \oplus A' \supset B \) means \( ((D \oplus A) \oplus A') \supset B \). Additionally, when referring to a finite series of consecutive unions such as \( A_1 \oplus \ldots \oplus A_n \), we will use the simplified notation \( \oplus_{i \in 1..n} A_i \). This notation is not strict on how subexpressions \( A_i \) are associated, hence, in principle, it refers to any of all possible associations. In the next section we present an equivalence relation on \( \mu \)-types that will identify all these associations. We often write \( \mu V.A \) to mean either \( \mu \alpha.D \) or \( \mu X.A \). A non-union \( \mu \)-type \( A \) is a \( \mu \)-type of one of the following forms: \( \alpha, \varepsilon, D \oplus A, X, A \supset B \) or \( \mu V.A \) with \( A \) a non-union \( \mu \)-type. We assume \( \mu \)-types are contractive: \( \mu V.A \) is contractive if \( V \) occurs in \( A \) only under a type constructor \( \supset \) or \( \oplus \), if at all. We henceforth redefine \( T \) to be the set of contractive \( \mu \)-types. \( \mu \)-types come equipped with a notion of equivalence \( \simeq_\mu \) and subtyping \( \preceq_\mu \).

**Definition 3.2**

(i) \( \simeq_\mu \) is the least congruence closed under the schemes in Fig. 1.

(ii) \( \preceq_\mu \) is defined in Fig. 2 where a subtyping context \( \Sigma \) is a set of assumptions over type variables of the form \( V \preceq_\mu W \) with \( V,W \in \mathcal{V} \).

(E-CONTR) actually encodes two rules, one for datatypes \( (\mu \alpha.D) \) and one for arbitrary types \( (\mu X.A) \). Likewise for (E-FOLD). The relation resulting from dropping (E-CONTR) \([3,6]\) is called weak type equivalence \([8]\) and is known to be too weak to capture equivalence of its coinductive formulation (required for our proof of invertibility of subtyping cf. Prop. 3.12); for example, types \( \mu X.A \supset A \supset X \) and \( \mu X.A \supset X \) cannot be equated. We can now use notation \( \oplus_{i \in 1..n} A_i \) on contractive \( \mu \)-types to denote several consecutive applications of the binary operator \( \oplus \) irrespective of how they are associated. All such associations yield equivalent \( \mu \)-types. Regarding the subtyping rules, we adopt those for union of \([26]\). It should be noted that the naïve variant of (s-REC) in which \( \Sigma \vdash \mu V.A \preceq_\mu \mu V.B \) is deduced from \( \Sigma \vdash A \preceq_\mu B \), is known to be unsound \([1]\). We often abbreviate \( \vdash A \preceq_\mu B \) as \( A \preceq_\mu B \).

### 3.2 Typing Schemes

A typing context \( \Gamma \) (or \( \theta \)) is a partial function from term variables to \( \mu \)-types; \( \Gamma(x) = A \) means that \( \Gamma \) maps \( x \) to \( A \). We have two typing judgments, one for patterns \( \theta \vdash p : A \) and one for terms \( \Gamma \vdash s : A \). Accordingly, we have two sets of typing rules: Fig. 3, top and bottom. We write \( \theta \triangleright_p p : A \) to indicate that the typing judgment \( \theta \vdash p : A \) is derivable (likewise for \( \Gamma \triangleright s : A \)). The typing schemes speak for themselves except for two of them which we now comment. The first is (T-APP). Note that we do not require the \( A_i \) to be non-union types. This allows examples
\[\Sigma \vdash A \preceq_{\mu} A \quad \Sigma, V \preceq_{\mu} W \vdash V \preceq_{\mu} W \quad \vdash A \simeq_{\mu} B \] (s-refl)\[\Sigma \vdash A \preceq_{\mu} B \quad \Sigma \vdash B \preceq_{\mu} C \quad \Sigma \vdash A \preceq_{\mu} C \] (s-trans)\[\Sigma \vdash A \preceq_{\mu} B' \quad \Sigma \vdash B \preceq_{\mu} B' \quad \Sigma \vdash A' \preceq_{\mu} B' \] (s-comp)\[\Sigma, V \preceq_{\mu} W \vdash A \preceq_{\mu} B \quad W \not\in \text{fv}(A) \quad V \not\in \text{fv}(B) \quad \Sigma \vdash \mu V.A \preceq_{\mu} \mu W.B \] (s-rec)

Fig. 2. Strong subtyping for \(\mu\)-types (Sample)

Patterns

\[
\begin{align*}
\theta(x) &= A \\
\theta \vdash_p x : A & \quad \text{(P-MATCH)} \\
\vdash_p c : c & \quad \text{(P-CONST)} \\
\vdash_p p : D & \quad \vdash_p q : A \quad \text{(P-COMP)}
\end{align*}
\]

Terms

\[
\begin{align*}
\Gamma(x) &= A & \quad \text{(T-VAR)} \\
\Gamma \vdash x : A & \quad \text{(T-APP)} \\
\Gamma \vdash c : c & \quad \text{(T-APP)} \\
\vdash r : D & \quad \vdash u : A & \quad \text{(T-COMP)} \\
\Gamma \vdash r : D & \quad \Gamma \vdash u : A \quad \Gamma \vdash r u : D @ A \quad \text{(T-COMP)} \\
\vdash r : D & \quad \vdash u : A \quad \vdash p : D @ A \quad \text{(T-SUBS)} \\
\vdash r : D & \quad \vdash u : A \quad \vdash s : A' \quad \text{(T-ABS)}
\end{align*}
\]

Fig. 3. Typing rules for patterns and terms

such as (5) to be typable (the outermost instance of (T-APP) is with \(n = 1\) and \(A_1 = \text{Bool} = \text{true} \oplus \text{false}\)). Regarding (T-ABS) it requests a number of conditions. First of all, each of the patterns \(p_i\) must be typable under the typing context \(\theta_i\), \(i \in 1..n\). Also, the set of free matchables in each \(p_i\) must be exactly the domain of \(\theta_i\). Another condition, indicated by \((\Gamma, \theta_i \vdash s_i : B)_{i \in 1..n}\), is that the bodies of
each of the branches \( s_i, \ i \in 1..n \), be typable under the context extended with the corresponding \( \theta_i \). More noteworthy is the condition that the list \( [p_i : A_i]_{i \in 1..n} \) be \textit{compatible}, which we now discuss in further detail.

3.3 Compatibility

Let us say that a pattern \( p \) subsumes a pattern \( q \), written \( p \preceq q \) if there exists a substitution \( \sigma \) s.t. \( \sigma p = q \). Consider an abstraction \( (p \rightarrow_{\theta} \ s \mid q \rightarrow_{\theta'} t) \) and two judgments \( \theta \vdash p : A \) and \( \theta' \vdash q : B \). We consider two cases depending on whether \( p \) subsumes \( q \) or not.

As already mentioned in example (3) of the introduction, if \( p \) subsumes \( q \), then the branch \( q \rightarrow_{\theta'} t \) will never be evaluated since the argument will already match \( p \). Indeed, for any term \( u \) of type \( B \) in matchable form, the application will reduce to \( \{ u/p \}\) s. Thus, in this case, in order to ensure SR we demand that \( B \preceq \mu A \).

Suppose \( p \) does not subsume \( q \) (i.e. \( p \not\preceq q \)). We analyze the cause of failure of subsumption in order to determine whether requirements on \( A \) and \( B \) must be put forward. In some cases no requirements are necessary. For example in:

\[
\begin{align*}
f \rightarrow_{\{f:A \supset B\}} & \ (c \ z \rightarrow_{\{z:A\}} c (f z)) \\
| & \ d \ y \rightarrow_{\{y:B\}} x y
\end{align*}
\]

no relation between \( A \) and \( B \) is required since the branches are mutually disjoint. In other cases, however, \( B \preceq \mu A \) is required; we seek to characterize them. We focus on those cases where \( p \) fails to subsume \( q \), and \( \pi \in \text{pos}(p) \cap \text{pos}(q) \) is an offending position in both patterns. The following table exhaustively lists them:

<table>
<thead>
<tr>
<th>( p\mid\pi )</th>
<th>( q\mid\pi )</th>
<th>\text{restriction required}</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>( y )</td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td>( c )</td>
<td>( d )</td>
</tr>
<tr>
<td>(c)</td>
<td>( q_1 \ q_2 )</td>
<td></td>
</tr>
<tr>
<td>(d)</td>
<td>( y )</td>
<td></td>
</tr>
<tr>
<td>(e)</td>
<td>( p_1 \ p_2 )</td>
<td>( d )</td>
</tr>
</tbody>
</table>

In cases (b), (c) and (e), no extra condition on the types of \( p \) and \( q \) is necessary either, since their respective sets of possible arguments are disjoint; example (6) corresponds to the first of these. The cases where \( A \) and \( B \) must be related are (a) and (d): for those we require \( B \preceq \mu A \). The first of these has already been illustrated in the introduction (3), the second one is illustrated as follows:

\[
\begin{align*}
f \rightarrow_{\{f:D \supset A \supset C\}} & \ g \rightarrow_{\{y:B \supset C\}} (x \ y \rightarrow_{\{x:D, y:A\}} f x y) \\
| & \ z \rightarrow_{\{z:B\}} g z
\end{align*}
\]

The problematic situation is when \( B = D' \ @ \ B', \) i.e. the type of \( z \) is another compound, which may have no relation at all with \( D \ @ \ A \). Compatibility ensures
We now formalize these ideas.

**Definition 3.3** Given a pattern $\theta \vdash p : A$ and $\pi \in \text{pos}(p)$, we say $A$ admits a symbol $\odot$ (with $\odot \in V \cup C \cup \{\supset, @\}$) at position $\pi$ iff $\odot \in A \parallel \pi$, where:

- $a \parallel \epsilon \triangleq \{a\}$
- $(A_1 \star A_2) \parallel \epsilon \triangleq \{\star\}$, $\star \in \{\supset, @\}$
- $(A_1 \star A_2) \parallel i \pi \triangleq A_1 \parallel \pi \cup A_2 \parallel \pi$
- $(\mu V.A' \parallel \pi \triangleq (\{\mu V.A'/V\} A') \parallel \pi$

Note that $\theta \vdash p : A$ and contractiveness of $A$, implies $A \parallel \pi$ is well-defined for $\pi \in \text{pos}(p)$.

Whenever subsumption between two patterns fails, any mismatching position is a leaf in the syntactic tree of one of the patterns. Otherwise, both of them would have a type application constructor in that position and there would be no failure of subsumption.

**Definition 3.4** The **maximal positions** in a set of positions $P$ are:

$$\text{maxpos}(P) \triangleq \{\pi \in P \mid \exists \pi' \in P. \pi' = \pi \pi'' \land \pi'' \neq \epsilon\}$$

The **mismatching positions** between two patterns are:

$$\text{mmpos}(p, q) \triangleq \{\pi \mid \pi \in \text{maxpos}(\text{pos}(p) \cap \text{pos}(q)) \land p|\pi \not\sqsubseteq q|\pi\}$$

**Definition 3.5** We say $p : A$ is **compatible** with $q : B$, written $p : A \ll q : B$, iff the following two conditions hold:

(i) $p \not\ll q \Rightarrow B \preceq A$

(ii) $p \not\ll q \Rightarrow (\forall \pi \in \text{mmpos}(p, q). A \parallel \pi \cap B \parallel \pi \neq \emptyset) \Rightarrow B \preceq A$

A list of patterns $[p_i : A_i]_{i \in 1..n}$ is compatible if $\forall i, j \in 1..n.i < j \Rightarrow p_i : A_i \ll p_j : A_j$.

As a further example, suppose we wish to apply $\text{upd}$ (cf. (1)) to data structures holding values of different types: say $v1$ prefixed values are numbers and $v12$ prefixed values are functions over numbers. Note that $\text{upd}$ cannot be typed as it stands. The reason is that the last branch would have to handle values of functional type and hence would receive type $\text{cons} \oplus \text{node} \oplus \text{nil} \oplus v1 \oplus \text{Nat} \supset v2 \oplus (\text{Nat} \supset \text{Nat})$. This fails to be a datatype due to the presence of the component of functional type. As a consequence, $xy$ cannot be typed since it requires an applicative type @. The remedy is to add
an additional branch to \( \text{upd} \) capable of handling values prefixed by \( \text{vl2} \):

\[
\text{upd}' = f \rightarrow (f;A \supset B) \quad g \rightarrow (g;(A_2 \supset A_3) \supset B) \quad (\text{vl1} \ z \rightarrow \{z;A_1\} \quad \text{vl1} \ (f \ z)) \\
| \quad \text{vl2} \ z \rightarrow \{z;A_2 \supset A_3\} \quad \text{vl2} \ (g \ z) \\
| \quad x \ y \rightarrow \{x;c;\{y;D\}\} \quad (\text{upd}' \ f \ x) \ (\text{upd}' \ f \ y) \\
| \quad w \rightarrow \{w;E\} \quad w)
\]

The type of \( \text{upd}' \) is \( (A_1 \supset B) \supset ((A_2 \supset A_3) \supset B) \supset (F_{A_1,A_2 \supset A_3} \supset F_{B,B}) \), where \( F_{X,Y} \) is

\[
\mu \alpha.(\text{vl} \ @ \ X) \oplus (\text{vl}2 \ @ \ Y) \oplus (\alpha \ @ \ \alpha) \oplus (\text{cons} \oplus \text{node} \oplus \text{nil})
\]

This is quite natural: the type system establishes a clear distinction between semi-structured data, susceptible to path polymorphism, and “unstructured” data represented here by base and functional types.

### 3.4 Basic Metatheory of Typing

We present some technical lemmas that will be useful in the proof of safety. This first lemma is a straightforward adaptation of the standard Generation Lemma to our system.

**Lemma 3.6 (Generation Lemma)** Let \( \Gamma \) be a typing context and \( A \) a type.

(i) If \( \Gamma \vdash x : A \) then \( \exists A' \text{ s.t. } A' \preceq \mu A \) and \( x : A' \in \Gamma \).

(ii) If \( \Gamma \vdash c : A \) then \( c \preceq \mu A \).

(iii) If \( \Gamma \vdash r \ u : A \) then:

(a) either \( \exists D, A' \text{ s.t. } D @ A' \preceq \mu A, \Gamma \vdash r : D \) and \( \Gamma \vdash u : A' \);

(b) or \( \exists A_1, \ldots, A_n, A', k \in 1..n \text{ s.t. } A' \preceq A, \Gamma \vdash r : \oplus_{i \in 1..n} A_i \supset A', \) and \( \Gamma \vdash u : A_k \).

(iv) If \( \Gamma \vdash (p_i \rightarrow_{\theta_i} s_i)_{i \in 1..n} : A \) then \( \exists A_1, \ldots, A_n, B \text{ s.t. } \oplus_{i \in 1..n} A_i \supset B \preceq \mu A \), \( [p_i : A_i]_{i \in 1..n} \) is compatible, \( \text{dom} (\theta_i) = \text{fm}(p_i) \), \( \theta_i \vdash p_i : A_i \) and \( \Gamma, \theta_i \vdash s_i : B \) for every \( i \in 1..n \).

The following lemma is useful to deduce the shape of the type when we know the term is a data structure. Essentially it states that every data structure that can be given a type, can also be typed with a more specific non-union datatype.

**Lemma 3.7 (Typing for Data Structures)** Suppose \( \Gamma \vdash d : A \), for \( d \) a data structure. Then \( \exists D \) datatype such that \( D \preceq \mu A \) and \( \Gamma \vdash d : D \). Moreover,

(i) If \( d = c \), then \( D \simeq \mu c \).

(ii) If \( d = d' \ t \), then \( \exists D', A' \text{ such that } D \simeq \mu D' \oplus A', \Gamma \vdash d' : D' \) and \( \Gamma \vdash t : A' \).

Some results on compatibility follow, the crucial one being Lem. 3.9. This next lemma shows that matching failure is enough to guarantee that the type of the argument is not a subtype of that of the pattern.
Lemma 3.8 Given $\Gamma \triangleright u : B$, $\theta \triangleright p : A$. If $\{u/p\} = \text{fail}$, then $B \not\preceq \mu A$.

Define $\mathcal{P}_{\text{comp}}(p : A, q : B) \triangleq p \triangleleft q \lor (\forall \pi \in \text{mmpos}(p, q). A\nmid\pi \cap B\nmid\pi \neq \emptyset)$ so that compatibility can alternatively be characterized as:

$$p_i : A_i \ll p_j : A_j \quad \text{iff} \quad \mathcal{P}_{\text{comp}}(p : A, q : B) \implies B \preceq \mu A$$

The Compatibility Lemma should be interpreted in the context of an abstraction. Assume an argument $u$ of type $B$ is passed to a function where there are (at least) two branches, defined by patterns $p$ and $q$, the latter having the same type as $u$. If the argument matches the first pattern of (potentially) a different type $A$, then $\mathcal{P}_{\text{comp}}(p : A, q : B)$ must hold. Since patterns within an abstraction must be compatible, we get $B \preceq \mu A$ and thus $\Gamma \triangleright u : A$ too.

Lemma 3.9 (Compatibility Lemma) Suppose $\Gamma \triangleright u : B$, $\theta \triangleright p : A$, $\theta' \triangleright q : B$ and $\{u/p\}$ is successful. Then, $\mathcal{P}_{\text{comp}}(p : A, q : B)$ holds.

We write $\Gamma \triangleright \sigma : \theta$ to indicate that $\text{dom} (\sigma) = \text{dom} (\theta)$ and $\Gamma \triangleright \sigma(x) : \theta(x)$, for all $x \in \text{dom} (\sigma)$.

The following lemma assures that the substitution yielded by a successful match preserves the types of the variables in the pattern.

Lemma 3.10 (Type of Successful Match) Suppose $\{u/p\} = \sigma$ is successful, $\text{dom} (\theta) = \text{fm}(p)$, $\theta \triangleright p : A$ and $\Gamma \triangleright u : A$. Then $\Gamma \triangleright \sigma : \theta$.

Finally, we recall to the standard Substitution Lemma for type systems. It may also be interpreted in the context of an abstraction. Given $p \rightarrow_s s$, where $\theta$ has the type assignments for the variables in $p$, every substitution that preserves $\theta$ will also preserve the type of $s$ once $\theta$ is abstracted.

Lemma 3.11 (Substitution Lemma) Suppose $\Gamma, \theta \triangleright s : A$ and $\Gamma \triangleright \sigma : \theta$. Then $\Gamma \triangleright \sigma s : A$.

Type safety, addressed in the next section, also relies on $\preceq_\mu$ enjoying the fundamental property of invertibility of non-union types:

Proposition 3.12 (i) If $D @ A \preceq_\mu D' @ A'$, then $D \preceq_\mu D'$ and $A \preceq_\mu A'$.

(ii) If $A \supset B \preceq_\mu A' \supset B'$, then $A' \preceq_\mu A$ and $B \preceq_\mu B'$.

To prove this we appeal to the standard tree interpretation of terms and formulate an equivalent coinductive definition of equivalence and subtyping.

For the latter, invertibility of non-union types is proved coinductively, entailing Prop. 3.12 (cf. [25]).

4 Safety

Subject Reduction (Prop. 4.1) and Progress (Prop. 4.2) are addressed next.

Proposition 4.1 (Subject Reduction) If $\Gamma \triangleright s : A$ and $s \rightarrow s'$, then $\Gamma \triangleright s' : A$. 

Viso, Bonelli, and Ayala-Rincon

Proof. By induction on $s$. The non-trivial case is when $s = (p_i \rightarrow \theta_i, s_i)_{i \in 1..n} u$ and $s' = \{u/p_k\} s_k$ for some $k \in 1..n$ such that $\{u/p_k\} = \sigma$ and $\{u/p_k\} = \text{fail}$ for every $i < k$. By Generation Lemma (iii.b), there exists $C_1, \ldots, C_m, A'$ such that $A' \preceq \mu A$, $\Gamma \triangleright r : \bigoplus_{j \in 1..m} C_j \preceq \mu A'$ and:

$$\Gamma \triangleright u : C_{k'}$$

4 for some $k' \in 1..m$. Applying once again the Generation Lemma, item (iv) this time, to $\Gamma \triangleright r : \bigoplus_{j \in 1..m} C_j \preceq \mu A'$, we get $\exists A_1, \ldots, A_n, B$ such that:

$$\bigoplus_{i \in 1..n} A_i \supset B \preceq \mu \bigoplus_{j \in 1..m} C_j \preceq \mu A'$$

From (10), by invertibility of subtyping for non-union types, we have $B \preceq \mu A'$ and

$$\bigoplus_{j \in 1..m} C_j \preceq \mu \bigoplus_{i \in 1..n} A_i$$

We want to show that $\Gamma \triangleright u : A_k$. For that we need to distinguish two cases:

(i) If $u$ is in matchable form, we have two possibilities:

(a) $u$ is a data structure: then, by the Typing for Data Structures lemma, there exists a non-union datatype $D$ such that $D \preceq \mu C_{k'}$ and $\Gamma \triangleright u : D$.

(b) $u$ is an abstraction: then, by Generation Lemma (iv), there exists types $C', C''$ such that $C' \supset C'' \preceq \mu C_{k'}$ and $\Gamma \triangleright u : C' \supset C''$.

Then, in both cases there exists a non-union type, say $C$, such that $C \preceq \mu C_{k'}$ and $\Gamma \triangleright u : C$. Then, from (11) we get:

$$C \preceq \mu \bigoplus_{i \in 1..n} A_i$$

and, since $C$ is non-union, $C \preceq \mu A_l$ for some $l \in 1..n$. Hence, by subsumption $\Gamma \triangleright u : A_l$.

If $k = l$ we are done, so assume $k \neq l$. Recall the conditions for the reduction rule, where $\{u/p_i\} = \text{fail}$ for every $i < k$. Then, by Lem. 3.8, we have $A_l \preceq \mu A_i$. Thus, it must be the case that $k < l$. By Lem. 3.9 with hypothesis $\Gamma \triangleright u : A_l$, $\theta_k \triangleright p_k : A_k$, $\theta_l \triangleright p_l : A_l$ and $\{u/p_k\} = \sigma$ we get that $P_{\text{comp}}(p_k : A_k, p_l : A_l)$ holds. Additionally, we already saw that the list $[p_i : A_i]_{i \in 1..n}$ is compatible, thus $p_k : A_k \ll p_l : A_l$ and by definition $A_l \preceq \mu A_k$. Finally, we conclude by subsumption once again, $\Gamma \triangleright u : A_k$.

(ii) If $u$ is not in matchable form, then $p_k = x$ and by the premises of the reductions rule we need $\{u/p_i\} = \text{fail}$ for every $i < k$. Thus, necessarily $k = 1$. Moreover, since $x \ll p_i$ for every $i \in 1..n$, by compatibility we have $A_i \preceq \mu A_k$.

Then, from (11) we get

$$C_{k'} \preceq \mu \bigoplus_{j \in 1..m} C_j \preceq \mu \bigoplus_{i \in 1..n} A_i \preceq \mu A_k$$

Thus, by subsumption, $\Gamma \triangleright u : A_k$. 

16
Finally, in either case we have $\Gamma \vdash u : A_k$. Now Lem. 3.10 and 3.11 with $\Gamma, \theta_k \vdash s_k : B$ entails $\Gamma \vdash s' : B$ and we conclude by subsumption, $\Gamma \vdash s' : A$ (recall $B \leq_{\mu} A' \leq_{\mu} A$).

Let the set of values be defined as $v ::= x v_1 \ldots v_n \mid c v_1 \ldots v_n \mid (p_{i} \rightarrow_{\theta_i} s_i)_{i \in 1..n}$. The following auxiliary property guarantees the success of matching for well-typed closed values (note that values are already in matchable form).

**Proposition 4.2 (Progress)** If $\vdash s : A$ and $s$ is not a value, then $\exists s' \text{ s.t. } s \rightarrow s'$.

The proof is by induction on the term analyzing those subterms that can still be reduced to a value. Full details are available in the complete report [25].

### 5 Conclusions

A type system is proposed for a calculus that supports path polymorphism and two fundamental properties are addressed, namely subject reduction and progress. The type system includes type application, constants as types, union and recursive types. Both properties rely crucially on a notion of pattern compatibility and on invertibility of subtyping of $\mu$-types. This last result is proved via a coinductive semantics for the finite $\mu$-types. Regarding future work an outline of possible avenues follows.

- There exists extensive work on type-checking for recursive types [1, 18, 24], including some efficient algorithms for both equivalence [21] and subtyping [19]. Some of this work may be adapted to get a decidability result for CAP which may lead to an implementation of the system.
- We already mentioned the addition of parametric polymorphism (presumably in the style of F$_{<}$: [7, 14, 23]). We believe this should not present major difficulties.
- Strong normalization requires devising a notion of positive/negative occurrence in the presence of strong $\mu$-type equality, which is known not to be obvious [4, page 515].
- A more ambitious extension is that of dynamic patterns, namely patterns that may be computed at run-time, PPC being the prime example of a calculus supporting this feature.

### References


Completeness in PVS of a Nominal Unification Algorithm

Mauricio Ayala-Rincón a,2 Maribel Fernández b,3 Ana Cristina Rocha-Oliveira a,1

a Departamentos de Matemática e Ciência da Computação
Universidade de Brasília
Brasília D.F., Brasil

b Department of Informatics
King’s College London
London, UK

Abstract

Nominal systems are an alternative approach for the treatment of variables in computational systems. In the nominal approach variable bindings are represented using techniques that are close to first-order logical techniques, instead of using a higher-order metalanguage. Functional nominal computation can be modelled through nominal rewriting, in which α-equivalence, nominal matching and nominal unification play an important role. Nominal unification was initially studied by Urban, Pitts and Gabbay and then formalised by Urban in the proof assistant Isabelle/HOL and by Kumar and Norrish in HOL4. In this work, we present a new specification of nominal unification in the language of PVS and a formalisation of its completeness. This formalisation is based on a natural notion of nominal α-equivalence, avoiding in this way the use of the intermediate auxiliary weak α-relation considered in previous formalisations. Also, in our specification, instead of applying simplification rules to unification and freshness constraints, we recursively build solutions for the original problem through a straightforward functional specification, obtaining a formalisation that is closer to algorithmic implementations. This is possible by the independence of freshness contexts guaranteed by a series of technical lemmas.

Keywords: Nominal terms, binders, α-equivalence, nominal unification, PVS.

1 Introduction

When one introduces variable binders in a language, one thing to be considered immediately is α-equivalence. For instance, it must be possible to derive the equivalence between the formulas ∃x : x > 1 and ∃y : y > 1, despite the syntactical differences. Nominal theories treat binders in a way that is closer to informal practice, using variable names and freshness constraints instead of using indices as in explicit substitutions à la de Bruijn. In nominal syntax, there are two kinds of variables: atoms, representing object-level variables, and meta-variables, or simply

1 Email: anacrimarie@gmail.com. Author supported by a Ph.D. scholarship from CAPES Brazil.
2 Email: ayala@unb.br. Work partially supported by grant CNPq UNIVERSAL 476952/2013-1.
3 Email: maribel.fernandez@kcl.ac.uk. Work partially supported by grant CsF PVE CAPES 146/2012.

Selected, revised papers will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
variables. Atoms can be abstracted but not substituted, whereas variables cannot be abstracted but can be substituted. The notion of substitution is first-order in the sense that it allows capture, but freshness constraints are taken into account. Notions such as rewriting (cf. [FG07]) and unification (cf. [UPG04]) can be directly defined, without having to rely on involved notions such as β-reduction, as in the higher-order and explicit substitutions approaches (cf. [Hue75, DHK00, ARK01]).

Nominal unification problems can be solved (modulo α-equivalence) with first-order substitutions that act over meta-variables, i.e., simply filling the holes marked with meta-variables (X, Y, Z, ...) and allowing capture of variable names (a, b, c, i, k, ...). This can be illustrated by the expressions

\[ \sum_{k=0}^{7} \sum_{i=0}^{5} (i - X)^{i} \quad \text{and} \quad \sum_{i=0}^{7} \sum_{k=0}^{5} (X - Y)^{k}, \]

which admit a most general unifier according to the algorithm in [UPG04], with solution \[ X \mapsto k \mid Y \mapsto i \]. Note that \( i \) and \( k \) are captured, because these names are bound or abstracted by the sum operator. In a higher-order unification approach, this solution would not be accepted because bound variable capture is forbidden.

On the other hand, the unification problem with the expressions

\[ \sum_{i=0}^{5} (i - X)^{i} \quad \text{and} \quad \sum_{k=0}^{5} (X - Y)^{k} \]

has no solution in the nominal setting. One could argue that a solution could be obtained instantiating \([X \mapsto i] [Y \mapsto i] \) and renaming \( k \) as \( i \). But this is not possible since \( i \) should be a “fresh” name in the scope of the second sum in order to proceed with this renaming, and the chosen substitution contradicts this condition. In other words, the meta-variable \( X \) should be instantiated uniformly among the problem.

Translations between nominal unification problems and higher-order pattern unification problems are given in [Che05, LV12].

1.1 Contribution

In this paper, we present a functional specification of a new nominal unification algorithm and formalise its correctness and completeness in the language of the higher-order proof assistant Prototype Verification System (PVS) [SORSC01]. PVS was chosen because it has a large library about term rewriting systems ([GAR10]) and our nominal unification theory extends this background about rewriting.

The style of our specification is close to the functional presentations of Robinson’s first-order unification algorithm, and the formalisation avoids the use of intermediate equivalence relations, obtaining in a straightforward manner transitivity and symmetry of the nominal α-equivalence relation. Indeed, in [Urb10b], a “weak equivalence” is used in order to simplify the proof of transitivity for the standard nominal α-equivalence. However, in this paper, we present an even simpler proof, avoiding formalisations of properties of this weak intermediate relation. This is obtained following the analytic scheme of proof shown in [FG07].
The nominal unification algorithm given in Isabelle/HOL in [Urb10a] is essentially specified as the transformation rule system presented in [UPG04]. These rules transform unification problems with their associated freshness contexts into simpler ones. This approach is very elegant and allows a higher level of abstraction that simplifies the analysis of computational properties such as termination and uniqueness of solutions, but it is not so useful in implementations due to its inherent non-determinism (regarding the application of the transformation rules).

Here we present a new nominal unification algorithm that has only two nominal terms (but no freshness context) as parameters, as done in [CF10,LV10]. However, the algorithms presented in [CF10,LV10] focus on efficiency, whereas our goal is to formalise the proof of correctness by specifying the algorithm in PVS as a recursive function “unify” working directly on terms and formalising separately properties of contexts. Although the function “unify” does not carry freshness contexts, it builds them at the end of the execution together with the substitution solution. The freshness problems generated during the recursive computation are solved separately due to the independence of solutions for freshness and without involving extra fresh variables as usual in a nominal setting. This differs from the treatment given in [LV10] where freshness constraints, as well as suspensions, are encoded as equations, that was proved equivalent to the treatment in [CF10] in [Cal13].

1.2 Related work

There are formalisations of nominal theories in other proof assistants. The most relevant formalisation has been implemented in Isabelle/HOL [Urb08], where α-equivalence between terms is effectively obtained by representing terms as “abstraction functions”. Thus, Urban [Urb08] presents some basic conditions that are sufficient to guarantee the equivalence between two representations of terms, then, an induction principle is presented, to obtain proofs by induction over abstracted terms in a more natural way. For instance, the Substitution Lemma (well-known in the context of λ-calculus) was formalised using these techniques.

A similar work was done in Coq [ABW07], but bound variables were encoded by using de Bruijn indices and the terms were defined as having the type of locally nameless terms. An induction principle was implemented in order to prove properties about well-formed terms without mentioning indices.

Another formalisation in Isabelle/HOL is available in [Urb10a], to deal with nominal unification following [UPG04]. This formalisation is closer to ours in the sense that α-equivalence is defined under some side-conditions (namely, freshness conditions). The properties formalised in this system include the fact that the specified α-equivalence is indeed an equivalence relation, termination and soundness of the unification algorithm and characterisation of the normal forms generated by the algorithm.

In [Urb10b], Urban compares the proof of transitivity of the α-equivalence relation presented in [Urb10a,FG07] and [KN10]. The proof shown in the last citation was then considered the best because it avoids a more complex inductive scheme on the size of terms. However, it requires the implementation of a “weak-equivalence” relation as a workaround. Here, we follow auxiliary lemmas developed in [FG07], but...
with a simpler proof of transitivity by induction on the structure of terms obtaining directly the necessary result that the specified $\alpha$-equivalence relation is indeed an equivalence relation.

The specification of our algorithm, passing as parameters only pairs of terms to be unified, is closer to functional presentations in the style of Robinson’s first-order unification that have been repeatedly formalised in a variety of proof assistants (e.g., [Pau85,AGdMA14]).

1.3 Organisation

Section 2 presents the basic concepts and grammars used in the nominal context. Section 3 defines freshness and $\alpha$-equivalence and makes explicit (subsection 3.1) the details about the proof of transitivity of $\alpha$-equivalence used in previous formalisations in comparison with the ones strictly necessary in the current approach. Also, this section (subsection 3.2) presents a function that computes the minimal freshness context needed to derive a freshness constraint. This is crucial for obtaining a unification algorithm that does not need to carry freshness contexts continuously. Section 4 presents the main contributions of this paper: the specification of a functional algorithm to solve nominal unification problems and the formalisation of its soundness and completeness.

This paper is accompanied with the whole PVS development for nominal unification, which includes specifications of all notions and definitions as well as formalisations of the proofs of all lemmas and theorems given in this paper, available for download in the PVS theory for term rewriting systems trs.cic.unb.br.

2 Preliminaries

This section presents some basic definitions: permutations, terms and substitutions, which are needed to reason about a nominal unification algorithm.

**Definition 2.1** Atoms or names are basic structures in the context of nominal theories. They represent object-level variables; the set $\mathcal{A}$ of all atoms is countably infinite. A swapping $(a\ b)$ is a bijection from $\mathcal{A}$ into $\mathcal{A}$ that exchanges $a$ and $b$ and that fixes any other atom. Permutations are also bijections of the form $\pi : \mathcal{A} \rightarrow \mathcal{A}$, which change a finite number of atoms and that are represented as lists of swappings. Then, the action of a permutation over atoms is recursively defined as:

$$id(c) = c,$$ where $id$ is the null list;

$$((a\ b) \circ \pi)(c) = \begin{cases} a, & \text{if } \pi(c) = b; \\ b, & \text{if } \pi(c) = a; \\ \pi(c), & \text{otherwise.} \end{cases}$$

The inverse of $\pi$ is the reverse list of swappings and it is denoted by $\pi^{-1}$.

**Definition 2.2** Let $\Sigma$ and $\mathcal{V}$ be a signature with function symbols and a countably infinite set of variables, respectively. Then, the set $\mathcal{T}(\Sigma,\mathcal{A},\mathcal{V})$ of nominal terms is generated by the following grammar:

$$t ::= \bar{a} \mid \pi \cdot X \mid () \mid (t_1,t_2) \mid [a]t \mid f\ t,$$
where $\bar{a}$ is an atom, $\pi \cdot X$ is a suspension (a permutation $\pi$ suspended in the variable $X \in V$), $()$ is the unit or empty tuple, $(t_1, t_2)$ is a pair of terms, $[a]t$ is an abstraction (a term with the atom $a$ abstracted) and $f t$ is an application (a symbol $f \in \Sigma$ applied to a term).

Notice that to encode terms in PVS, we distinguish between the atom $a$ and the term $\bar{a}$ that consists of the atom $a$ (compare with the constructor $at$ in the following code of the data structure of terms). Also, the function application works for symbols with arity one. To represent a greater arity, one can use pairs to encode tuples with any number of arguments. For instance, if the symbol $f$ has arity 3, then we can describe the term $f(t_1, (t_2, t_3))$ using the present grammar. The next specification of terms in PVS allows us to have induction schemes generated automatically.

\begin{verbatim}
BEGIN
  at (a: atom): atom?
  * (p: perm, V: variable): susp?
  unit: unit?
  pair (term1: term, term2: term): pair?
  abs (abstr: atom, body: term): abs?
  app (sym: symbol, arg: term): app?
END term
\end{verbatim}

**Definition 2.3** The depth of a term is computed by the following function:

$$
depth(\bar{a}) = depth(\pi \cdot X) = depth(() ) = 0 \quad depth([a]t) = 1 + depth(t)$$

$$
depth((t_1, t_2)) = 1 + \max(depth(t_1), depth(t_2)) \quad depth(f t) = 1 + depth(t)
$$

The function $depth$ is used as part of the measure provided to ensure termination of the nominal unification algorithm.

Actions of permutations can be homomorphically extended over terms. This means that permutations only change atoms and are accumulated into suspensions. A precise definition is given below.

**Definition 2.4** The action of a permutation $\pi$ over terms is defined as:

$$
\pi \cdot \bar{a} = \overline{\pi(a)} \quad \pi \cdot (\pi' \cdot X) = (\pi \circ \pi') \cdot X \quad \pi \cdot () = ()
$$

$$
\pi \cdot (t_1, t_2) = (\pi \cdot t_1, \pi \cdot t_2) \quad \pi \cdot [a]t = [\pi(a)]\pi \cdot t \quad \pi \cdot f t = f \pi \cdot t
$$

One important observation is that the variables in suspensions work as metavariables, where a substitution that replaces variables by terms is a primitive notion. With this in mind, it is reasonable that nominal variables are not abstractable. The denomination ‘suspension’ for $\bar{\pi} \cdot X$ has to do with the fact that the permutation $\pi$ cannot indeed apply to $X$ until the instance of this variable is known; so it is suspended.

In PVS, permutations are specified as lists of pairs of atoms. The function $\text{act}$ applies a permutation to an atom by the recursive action of the swappings that represent the permutation. On the other hand, the function $\text{ext}$ extends the action of permutations to terms homomorphically, i.e., it applies $\text{act}$ to atoms and accumulates permutations in suspensions.
- perm: TYPE = list[[atom,atom]]
- act(pi:perm)(c): RECURSIVE atom =
  CASES pi OF
  null: c,
  cons((a,b),rest): LET d = act(rest)(c) IN
  IF d = a THEN b
  ELSIF d = b THEN a
  ELSE d
  ENDIF
ENDCASES
MEASURE pi BY <<
- ext(pi:perm)(t:term): RECURSIVE term =
  CASES t OF
  at(a): at(act(pi)(a)),
  *(pm, v): *(append(pi, pm), v),
  unit: unit,
  pair(t1,t2): pair(ext(pi)(t1),ext(pi)(t2)),
  abs(ab, bd): abs(act(pi)(ab), ext(pi)(bd)),
  app(sl, ag): app(sl, ext(pi)(ag))
ENDCASES
MEASURE t BY <<

Remark 2.5 The necessity and use of ‘measure’ functions in PVS recursive functions is for proving termination according to the operational semantics of termination of PVS. This measure on the parameters should decrease after each recursive call. In the previous functions the measure ‘<<’ represents the standard measure on the data structures of permutations and terms; respectively, length of lists and the subterm relation. In some cases, as for these functions, the system can automatically verify the decrement of the measure provided.

Definition 2.6 A nuclear substitution is a pair of the form [X → s], where X is a variable and s is a term, and its action over terms is defined as:

\[
\begin{align*}
\bar{a}[X \mapsto s] &= \bar{a} \\
[a][t][X \mapsto s] &= [a][t[X \mapsto s]] \\
(\pi \cdot Y)[X \mapsto s] &= \begin{cases} 
\pi \cdot Y, & \text{if } X \neq Y \\
\pi \cdot s, & \text{otherwise}
\end{cases} \\
() [X \mapsto s] &= () \\
(f \ t)[X \mapsto s] &= f(t[X \mapsto s]) \\
(t_1,t_2)[X \mapsto s] &= (t_1[X \mapsto s], t_2[X \mapsto s])
\end{align*}
\]

A substitution \( \sigma \) is a list of nuclear substitutions, which are applied one-by-one over terms, i.e:

\[
t \ Id = t, \text{ where } Id \text{ is the empty list; } \quad t(\sigma \circ [X \mapsto s]) = (t \sigma)[X \mapsto s].
\]

Notation: If \( \sigma \) and \( \gamma \) are two substitutions, then \( \sigma \gamma \) represents the composition of such substitutions, i.e., \( \sigma \circ \gamma \).

Remark 2.7 This notion of substitution is different from the simultaneous application of nuclear substitutions. This approach is closer to triangular substitutions as explored in [KN10], with the view to be more space efficient.

Definition 2.8 The set of variables of a term is recursively computed by the function \( \text{Vars} \), as follows.

\[
\begin{align*}
\text{Vars}(\bar{a}) &= \emptyset \\
\text{Vars}(\pi \cdot X) &= \{X\} \\
\text{Vars}(()) &= \emptyset \\
\text{Vars}((t_1,t_2)) &= \text{Vars}(t_1) \cup \text{Vars}(t_2) \\
\text{Vars}([a]t) &= \text{Vars}(t) \\
\text{Vars}(f \ t) &= \text{Vars}(t)
\end{align*}
\]
The next lemma states the invariance of alternating the application of a permutation and a substitution on a term.

**Lemma 2.9** For any term \( t \), \( \pi \circ (t \sigma) = (\pi \circ t)\sigma \).

**Proof.** By induction on the structure of \( t \).

## 3 Freshness and \( \alpha \)-equivalence

As mentioned earlier, [UPG04] presented an algorithm to decide \( \alpha \)-equivalence of nominal terms, based on a notion of freshness of names in terms, without the necessity of generating new names.

**Definition 3.1** (Freshness) A **freshness context** \( \nabla \) is a finite set of pairs of the form \((a, X)\). We say that an atom \( a \) is **fresh** in \( t \) under \( \nabla \) (denoted by \( \nabla \vdash a \# t \)) if it is possible to build a proof of this judgement using the rules:

\[
\begin{array}{c}
\frac{}{\nabla \vdash a \# b} \quad \ \text{(#ab)} \\
\frac{}{\nabla \vdash a \# ( )} \quad \ \text{(#unit)} \\
\frac{\nabla \vdash a \# s_1 \quad \nabla \vdash a \# s_2}{\nabla \vdash a \# (s_1, s_2)} \quad \ \text{(#pair)} \\
\frac{}{\nabla \vdash a \# s} \quad \ \text{(#absb)} \\
\frac{}{\nabla \vdash a \# f s} \quad \ \text{(#f)}
\end{array}
\]

**Notation:** If \( \nabla \) and \( \Delta \) are freshness contexts, then \( \nabla \vdash \Delta \) means that \( \Delta \subseteq \nabla \) and \( \nabla \Delta \) denotes \( \nabla \cup \Delta \).

The following two auxiliary lemmas express invariance of derivability in the previous calculus under the action of permutations and weakening of freshness contexts.

**Lemma 3.2** \( \nabla \vdash a \# t \leftrightarrow \nabla \vdash \pi \cdot a \# \pi \cdot t \).

**Lemma 3.3** If \( \nabla \vdash \Delta \) and \( \Delta \vdash a \# t \), then \( \nabla \vdash a \# t \).

The proofs are by induction on the derivation of \( \Delta \vdash a \# t \).

Now, with the notions of permutation and freshness, \( \alpha \)-equivalence can be defined in a formal way.

**Definition 3.4** (**\( \alpha \)-equivalence**) The terms \( t \) and \( s \) are **\( \alpha \)-equivalent** in the context \( \nabla \), denoted by \( \nabla \vdash t \approx \alpha s \), if there is a proof of this judgement using the rules:

\[
\begin{array}{c}
\frac{}{\nabla \vdash a \approx \alpha \bar{a}} \quad \text{(**\( \approx \alpha, a \)**)} \\
\frac{}{\nabla \vdash ds(\pi, \pi') \# X \subseteq \nabla} \quad \text{(**\( \approx \alpha, X \)**)} \\
\frac{\nabla \vdash \pi \cdot \pi' \cdot X \approx \alpha \pi' \cdot X}{\nabla \vdash ( ) \approx \alpha ( )} \quad \text{(**\( \approx \alpha, O \)**)} \\
\frac{\nabla \vdash s_1 \approx a t_1 \quad \nabla \vdash s_2 \approx a t_2}{\nabla \vdash (s_1, s_2) \approx \alpha (t_1, t_2)} \quad \text{(**\( \approx \alpha, \text{pair} \)**)} \\
\frac{\nabla \vdash s \approx \alpha (a b) \cdot t \quad \nabla \vdash a \# t}{\nabla \vdash [a]s \approx \alpha [b]t} \quad \text{(**\( \approx \alpha, \text{absb} \)**)} \\
\frac{\nabla \vdash [a]s \approx \alpha [a]t}{} \quad \text{(**\( \approx \alpha, \text{abs} \)**)} \\
\frac{\nabla \vdash s \approx \alpha t \quad \nabla \vdash f s \approx \alpha f t}{\nabla \vdash s \approx \alpha f t} \quad \text{(**\( \approx \alpha, f \)**)}
\end{array}
\]
where \( ds(\pi, \pi') = \{ b \in A \mid \pi(b) \neq \pi'(b) \} \) (namely, the difference set between two permutations) and \( ds(\pi, \pi') \# X \) is the context formed by the pairs \((b, X)\), for each \( b \in ds(\pi, \pi')\).

### 3.1 A direct formalisation of transitivity of \( \alpha \)-equivalence

The next four auxiliary lemmas relate \( \alpha \)-equivalence, freshness and the action of permutations. The first one expresses preservation of freshness by \( \alpha \)-equivalent terms; the second one, alternation of the action of a permutation and its inverse on \( \alpha \)-equivalent terms; the third one, invariance of \( \alpha \)-equivalence under the action of a permutation; and, the fourth one, preservation of \( \alpha \)-equivalence of a term under the action of permutations whose difference set is fresh in the term.

**Lemma 3.5** \( \nabla \vdash a \# s \) and \( \nabla \vdash s \approx_\alpha t \) implies \( \nabla \vdash a \# t \).

**Lemma 3.6** \( \nabla \vdash s \approx_\alpha \pi \cdot t \Rightarrow \nabla \vdash \pi^{-1} \cdot s \approx_\alpha t \).

**Lemma 3.7** \( \nabla \vdash s \approx_\alpha t \Leftrightarrow \nabla \vdash \pi \cdot s \approx_\alpha \pi \cdot t \).

**Lemma 3.8** \( \nabla \vdash ds(\pi_1, \pi_2) \# t \) implies \( \nabla \vdash \pi_1 \cdot t \approx_\alpha \pi_2 \cdot t \).

Lemmas 3.5-3.8 are proved by induction on \( s \), applying Lemma 3.2. For Lemma 3.6, Lemma 3.5 is applied. The treatment is the same as in previous papers ([UPG04, FG07, Urb10b]) and their complete formalisations are available in the accompanying PVS development.

The proof of the next lemma is shown in detail because, at this point, the formalisation differs from the one given in [Urb10a] and reported in [Urb10b].

**Lemma 3.9** (Transitivity of \( \alpha \)-equivalence) The relation \( \approx_\alpha \) is transitive under a given context \( \nabla \), i.e., \( \nabla \vdash t_1 \approx_\alpha t_2 \) and \( \nabla \vdash t_2 \approx_\alpha t_3 \) implies \( \nabla \vdash t_1 \approx_\alpha t_3 \).

**Proof.** The proof is by induction on the structure of \( t_1 \).

- \( t_1 = \tilde{a} \): then by definition of \( \approx_\alpha \), \( t_2 = t_3 = \tilde{a} \).
- \( t_1 = \pi_1 \cdot X \); so \( t_2 = \pi_2 \cdot X \) and \( t_3 = \pi_3 \cdot X \). We need to prove that \( ds(\pi_1, \pi_3) \# X \subseteq \nabla \). So, take \( c \) such that \( \pi_1 \cdot c \neq \pi_3 \cdot c \). There are two cases: if \( \pi_1 \cdot c = \pi_2 \cdot c \), then \( \pi_2 \cdot c \neq \pi_3 \cdot c \) and \( (c, X) \in \nabla \) for \( ds(\pi_2, \pi_3) \# X \subseteq \nabla \); if \( \pi_1 \cdot c \neq \pi_2 \cdot c \), then \( (c, X) \in \nabla \) because \( ds(\pi_1, \pi_2) \# X \subseteq \nabla \).
- \( t_1 = () \) implies \( t_2 = () \) and \( t_3 = () \).
- \( t_1 = (s_1, s_2) \): then \( t_2 = (u_1, u_2) \) and \( t_3 = (w_1, w_2) \). By induction hypothesis, \( \nabla \vdash s_1 \approx_\alpha w_1 \) and \( \nabla \vdash s_2 \approx_\alpha w_2 \).
- \( t_1 = f \cdot s \): then \( t_2 = f \cdot u \) and \( t_3 = f \cdot w \). By induction hypothesis, \( \nabla \vdash s \approx_\alpha w \).
- \( t_1 = [a]s \): then \( t_2 = [b]u \) and \( t_3 = [c]w \). It is necessary to compare the abstractors:
  - \( a = b = c \): thus the result follows by induction hypothesis trivially.
  - \( a = b \neq c \): by definition, \( \nabla \vdash s \approx_\alpha u \) and \( \nabla \vdash u \approx_\alpha (b \cdot c) \cdot w \) and \( \nabla \vdash b \# w \). By IH, \( \nabla \vdash s \approx_\alpha (b \cdot c) \cdot w \). As \( a = b \), then freshness condition is satisfied to \( a \) as well.
  - \( a \neq b = c \): we have \( \nabla \vdash a \# u \), \( \nabla \vdash s \approx_\alpha (a \cdot c) \cdot u \) and \( \nabla \vdash u \approx_\alpha w \). By Lemma 3.7, \( \nabla \vdash (a \cdot c) \cdot u \approx_\alpha (a \cdot c) \cdot w \) and, by IH, \( \nabla \vdash s \approx_\alpha (a \cdot c) \cdot w \). By
Lemma 3.2, \(\nabla \vdash c\#(a\ c) \bullet u\) and \(\nabla \vdash c\#(a\ c) \bullet w\) by Lemma 3.5. Finally, again by Lemma 3.2, \(\nabla \vdash a\#w\).

- \(b \neq a = c\): it is known that \(\nabla \vdash s \approx_{\alpha} (b\ c) \bullet u\) and \(\nabla \vdash u \approx_{\alpha} (b\ c) \bullet w\). Then, \(\nabla \vdash (b\ c) \bullet u \approx_{\alpha} w\) by Lemma 3.6. By IH, \(\nabla \vdash s \approx_{\alpha} w\).

- \(a \neq b \neq c \neq a\): it is necessary to prove that \(\nabla \vdash s \approx_{\alpha} (a\ c) \bullet w\) and \(\nabla \vdash a\#w\). Let us prove first freshness: by definition of \(\approx_{\alpha}\), \(\nabla \vdash a\#u\) and \(\nabla \vdash u \approx_{\alpha} (b\ c) \bullet w\). By Lemma 3.5, \(\nabla \vdash a\#(b\ c) \bullet w\) and, by Lemma 3.2\((\approx)\), \(\nabla \vdash a\#w\). Now let us prove \(\alpha\)-equivalence: by hypothesis, \(\nabla \vdash s \approx_{\alpha} (a\ b) \bullet u\), \(\nabla \vdash u \approx_{\alpha} (c\ b) \bullet w\) and \(\nabla \vdash b\#w\). By Lemma 3.7, \(\nabla \vdash (a\ b) \bullet u \approx_{\alpha} (a\ b)(b\ c) \bullet w\). As \(ds((a\ b)(b\ c),(a\ c)) = \{a,b\}\) and both atoms are fresh in \(w\), then \(\nabla \vdash (a\ b)(b\ c) \bullet w \approx_{\alpha} (a\ c) \bullet w\) by Lemma 3.8. Now, applying IH twice, one obtains \(\nabla \vdash s \approx_{\alpha} (a\ c) \bullet w\).

\[\square\]

Note that the critical point in this proof is the abstraction, particularly when all the abstractors differ. This is due to the asymmetry of rule \((\approx_{\alpha}\text{abs})\) in Definition 3.4. The previous lemma was also presented in [UPG04,FG07], but in [Urb10b], a weak equivalence notion (Definition 3.10) is used as an intermediate relation to contour the problem with the abstraction case. However, auxiliary lemmas similar to the ones presented here were necessary in [Urb10b], beyond of other technical results to deal specifically with this weak equivalence (some of those additional lemmas in [Urb10b] are particular cases of transitivity). In the current formalisation, weak equivalence was not needed and the abstractions were treated as given in the five cases in the proof of Lemma 3.9.

**Definition 3.10 (Weak-equivalence)** Given two terms \(s, t\), they are said to be **weak equivalent** (notation: \(s \sim t\)) whenever there exists a derivation of it from the following rules:

\[
\begin{array}{|c|c|}
\hline
\text{Rule} & \text{Description} \\
\hline
\frac{\quad}{a \sim \bar{a}} & (\sim a) \\
\frac{s_1 \sim t_1}{s_1, s_2 \sim t_2} & (\sim \text{pair}) \\
\frac{s \sim t}{[a]s \sim [a]t} & (\sim \text{abs}) \\
\frac{ds(\pi, \pi') = \emptyset}{\pi \cdot X \sim \pi' \cdot X} & (\sim X) \\
\frac{s \sim t}{f \sim f} & (\sim f) \\
\hline
\end{array}
\]

In the previous definition, observe that when \(s \sim t\), then \(s\) and \(t\) differ only in possible representations of permutations \(\pi\) and \(\pi'\) in suspended variables. Even so, the action of those permutations must be equal. Thus, the relation \(\sim\) actually is closer to syntactic equality than to \(\alpha\)-equivalence. To obtain transitivity of \(\approx_{\alpha}\) using this definition, several auxiliary steps are necessary, among others, proving that \(\sim\) is invariant under the action of permutations, preservation of freshness by weak-equivalent terms, etc. These lemmas are similar to the previously mentioned for \(\approx_{\alpha}\). In addition, it is necessary to prove that, under a freshness context \(\Delta\), \((\approx_{\alpha} \circ \sim) \subseteq \approx_{\alpha}\), which is the key property for concluding transitivity of \(\approx_{\alpha}\). All this work is unnecessary in our approach.

**Lemma 3.11 (Equivalence)** \(\approx_{\alpha}\) is an equivalence relation under any context \(\nabla\).

**Proof.** Transitivity is guaranteed by Lemma 3.9. Reflexivity (\(\nabla \vdash t \approx_{\alpha} t\)) and
symmetry (∇ ⊢ t ≅α s implies ∇ ⊢ s ≅α t) are easy to verify through an inductive proof on the structure of t. The interesting case is the proof of symmetry for abstractions with different abstractors. In this case, ∇ ⊢ [a]t′ ≅α [b]s′ means ∇ ⊢ t′ ≅α (a b) • s′ and ∇ ⊢ a#s′. Applying (a b) to the freshness, we obtain ∇ ⊢ b#((a b) • s′) and, by Lemma 3.5, ∇ ⊢ b#t′. Now, by induction hypothesis, ∇ ⊢ (a b) • s′ ≅α t′ and, by Lemma 3.6, ∇ ⊢ s′ ≅α (a b) • t′. This proves ∇ ⊢ [b]s′ ≅α [a]t′. □

Notice that, unlike the proofs given in [UPG04, Urb10b], this formalised proof of symmetry does not use transitivity. Thus, these two properties are somehow independent from each other.

### 3.2 Minimal Freshness Contexts

As it will be shown in Section 4, a solution for a unification problem is a pair (∇, σ) of a freshness context and a substitution. What is expected from a nominal unification algorithm is that it generates “most general solutions” with respect to an ordering “≤” as in the first-order case (see Definition 4.11). In this way, in the current formalisation, a function was specified that can compute a minimal freshness context ∇ which derives a freshness problem a#t when possible, i.e., ∇ ⊢ a#t and ∇ is a subset of any other context ∆ such that ∆ ⊢ a#t.

In the next function, the measure “<<” denotes the proper subterm relation that is generated by PVS when the abstract data structure specified for terms is type-checked. As for the example in Remark 2.5, termination with respect to this measure can be automatically verified.

**Definition 3.12** Let a be an atom and t be a term. Define the function \((\# sol)\) that takes as input a pair \((a, t)\) and outputs a freshness context and a Boolean, as follows:

\[
(a#t)_{sol} := \text{CASES OF } t : \\
\quad b : (\varnothing, a \neq b), \\
\quad \pi \cdot X : ((\sigma^{-1} \cdot a, X), \text{True}), \\
\quad () : (\varnothing, \text{True}), \\
\quad (t_1, t_2) : \text{LET } (\Delta_1, b_1) = (a#t_1)_{sol}, (\Delta_2, b_2) = (a#t_2)_{sol} \text{ IN IF } b_1 = b_2 = \text{True THEN } (\Delta_1,\Delta_2, \text{True}) \text{ ELSE } (\varnothing, \text{False}), \\
\text{f } t : (a#t)_{sol}
\]

**MEASURE <<**

The function above was taken from the transformation rules related to the unification algorithm in [UPG04]. The difference is that here the freshness solutions are obtained separately from the substitutions which solve the equational problems in the unification algorithm. in this way, it is clear that the freshness constraints can restrict the validity of a unification problem, but they cannot modify the substitution that solves the problem.

The following lemma formalises the correctness of the previous definition.

**Lemma 3.13 (Correctness of \((\# sol)\))** Take \((\Delta, b) = (a#t)_{sol}\). Then,

\((b = \text{True } \Rightarrow \Delta \vdash a#t) \text{ and } (\text{for any } \nabla, \nabla \vdash a#t \Rightarrow b = \text{True and } \nabla \vdash \Delta).\)
Proof. The proof is by induction on the structure of $t$. The interesting case is when $t = (t_1, t_2)$, because to use rule (#pair), we need to have the same context in the derivations $\nabla \vdash a \# t_1$ and $\nabla \vdash a \# t_2$. However, the function $\langle \# \rangle_{\text{sol}}$ returns minimal contexts $\Delta_1$ and $\Delta_2$ to $t_1$ and $t_2$, respectively. For this reason, $\Delta_1$ and $\Delta_2$ have to be joined when computing $\langle \# \rangle_{\text{sol}}$. Then, using Lemma 3.3, it is possible to enlarge the contexts into the derivations $\Delta_1 \Delta_2 \vdash a \# t_1$ and $\Delta_1 \Delta_2 \vdash a \# t_2$ in order to be able to use the mentioned rule.

This function is crucial to build independently a freshness context for a whole nominal unification problem from its partial solutions, and it is used in the recursive treatment for the case of abstractions and pairs as will be explained in the next section.

Notation: $\langle \nabla \sigma \rangle_{\text{sol}}$ represents the union of solutions of $\langle a \# (id \cdot X) \sigma \rangle_{\text{sol}}$, for all $(a, X) \in \nabla$, with $\nabla \sigma$ denoting the resulting context if every subproblem is consistent, or $(\emptyset, False)$ if there is some inconsistency. The notation $\Delta \vdash \nabla \sigma$ states that $\Delta \vdash a \# (id \cdot X) \sigma$ is derivable for all $(a, X) \in \nabla$.

4 Nominal unification algorithm

In order to construct a nominal unification algorithm as a recursive function in the specification language of PVS, it is necessary to provide a recognisable answer in cases of failure, because PVS does not allow partial functions. To deal with failure, our algorithm will return triplets of the form $(\nabla, \sigma, b)$, which are a context, a substitution and a Boolean, respectively, instead of pairs of the form $(\nabla, \sigma)$. The triplet of the form $(\emptyset, Id, False)$ identifies failure cases and triplets of the form $(\nabla, \sigma, True)$ successful cases with solutions of the form $(\nabla, \sigma)$.

Definition 4.1 (Unifiable terms and unifiers) Two terms $t, s$ are said to be unifiable if there exists a context $\nabla$ and a substitution $\sigma$ such that $\nabla \vdash t \sigma =_\alpha s \sigma$. Under these conditions, the pair $(\nabla, \sigma)$ is called a unifier of $t$ and $s$.

Definition 4.2 (Nominal Unification Function) Let $t, s$ be two nominal terms. Then, we define the function

$$\text{unify}(t, s) := IF s = \pi \cdot X, AND X \notin \text{Vars}(t) \ THEN (\emptyset, [X \mapsto \pi^{-1} \cdot t], True)$$

$$ELSE$$

CASES OF $(t, s)$ :

$(\pi_1 \cdot X, \pi_2 \cdot X)$ : $(ds(\pi_1, \pi_2) \# X, Id, True)$,

$(\pi_1 \cdot X_1, s)$ : $IF X_1 \notin \text{Vars}(s) \ THEN (\emptyset, [X_1 \mapsto \pi_1^{-1} \cdot s], True)$,

$(\bar{a}, \bar{a}) : (\emptyset, Id, True)$,

$((a, s)) : (\emptyset, Id, True)$,

$((t_1, t_2), (s_1, s_2)) : \text{LET} (\nabla_1, \sigma_1, b_1) = \text{unify}(t_1, s_1)$,

$(\nabla_2, \sigma_2, b_2) = \text{unify}(t_2, s_1 \cdot 1, s_2)$,

$(\nabla_3, b_3) = (\nabla_1 \# \sigma_2)_{\text{sol}}$

$\text{IN} (\nabla_2 \nabla_3, \sigma_1 \sigma_2, b_1 \land b_2 \land b_3)$,

$([a] \tilde{t}, [b] \tilde{s}) : IF a = b \ THEN \text{unify}(\tilde{t}, \tilde{s})$

ELSE LET $(\nabla_1, \sigma, b_1) = \text{unify}(\tilde{t}, (a \cdot b) \cdot \tilde{s})$,

$(\nabla_2, b_2) = (a \# b)_{\text{sol}}$

$\text{IN} (\nabla_1 \nabla_2, \sigma, b_1 \land b_2)$,

$(f \tilde{t}, f \tilde{s}) : \text{unify}(\tilde{t}, \tilde{s})$,

ELSE : $(\emptyset, Id, False)$

$\text{MEASURE} \ lex(|\text{Vars}(t, s)|, \text{depth}(t))$
The measure function provided (see Remark 2.5) is lexicographic, with first component the number of variables in the unification problem and second component the depth of the first term of the unification problem.

The next remarks explain how the function $\langle \#, \rangle_{\text{sol}}$ correctly builds the necessary contexts for the abstraction and pair cases avoiding passing as parameter the freshness contexts, as done in unification mechanisms based on transformation rules (cf. [Urb10a]). In these remarks, unifiable terms are considered.

**Remark 4.3** In case of pairs, $(\nabla_2 \nabla_3, \sigma_1 \sigma_2)$ has to be a unifier for $(t_1, t_2)$ and $(s_1, s_2)$, i.e., $\nabla_2 \nabla_3 \vdash t_1 \sigma_1 \sigma_2 \approx_{\alpha} s_1 \sigma_1 \sigma_2$ and $\nabla_2 \nabla_3 \vdash t_2 \sigma_2 \approx_{\alpha} s_2 \sigma_1 \sigma_2$. Initially, unify builds the unifier $(\nabla_1, \sigma_1)$ for $t_1$ and $s_1$. Afterwards, $(\nabla_2, \sigma_2)$ is computed as a unifier for $t_2 \sigma_1$ and $s_2 \sigma_1$. Thus, $\nabla_1 \vdash t_1 \sigma_1 \approx_{\alpha} s_1 \sigma_1$ implies $\nabla_1 \nabla_2 \vdash t_1 \sigma_1 \sigma_2 \approx_{\alpha} s_1 \sigma_1 \sigma_2$ whenever $\nabla_3 = \nabla_1 \sigma_2$ is consistent. This can actually be solved through the function $\langle \#, \rangle_{\text{sol}}$, which has been proved to be sound and complete. Finally, since $\nabla_2 \vdash t_2 \sigma_1 \sigma_2 \approx_{\alpha} s_2 \sigma_1 \sigma_2$, weakening the contexts we obtain the desired unifier.

**Remark 4.4** When unifying two abstractions with different abstractors, the answer $(\nabla_1 \nabla_2, \sigma)$ has to be a unifier for $[a] t$ and $[b] s$. Indeed, initially the recursive call $\text{unify}(t, (a b) \bullet s)$ provides a unifier $(\nabla_1, \sigma)$ for this problem, if it is possible. Hence, $\nabla_1 \vdash t \sigma \approx_{\alpha} (a b) \bullet s \sigma$, but not necessarily $\nabla_1$ would be able to derive $a \# s \sigma$. Then, $\langle \#, \rangle_{\text{sol}}$ computes the minimal context $\nabla_2$ which derives $a \# s \sigma$ separately. Joining both contexts, the derivation $\nabla_1 \nabla_2 \vdash [a] t \sigma \approx_{\alpha} [b] s \sigma$ can be completed.

**Example 4.5** Take the problem of unifying $(X, X)$ and $((a b) \cdot X, a)$. First, one unifies $X$ and $(a b) \cdot X$. The result is the substitution $\text{Id}$ and the context $\{a \# X, b \# X\}$. Then, to unify $X \text{Id}$ and $a \text{Id}$, we need the substitution $[X \rightarrow a]$ and the empty context $\emptyset$. Then, $\{a \# X, b \# X\}$ is updated with $[X \rightarrow a]$, and $(a \# a)_{\text{sol}}$ returns failure.

Formalisation of termination of the function $\text{unify}$ is not obtained automatically and requires human intervention to show that $\text{lex}([\text{Vars}(t, s)], \text{depth}(t))$ decreases in each recursive call. Observe that there are recursive calls in the cases of pairs, abstractions and applications. In the last two cases one advances on the structure of the first (and second) terms calling recursively a problem with the same number of variables, but smaller depth. The same happens for the first recursive call in the case of pairs. For the second recursive call of the case of pairs, when $\text{unify}(t_2 \sigma_1, s_2 \sigma_1)$ is computed, if $\sigma_1 \neq \text{Id}$, the number of variables in the problem decreases for the nature of the nuclear substitutions generated in suspensions. So it is necessary to prove that the substitutions generated by $\text{unify}$ have a special characterisation, as explained in the next lemma.

**Definition 4.6** (Type $\text{Subs}(s)$ substitutions) The substitution $[X_1 \rightarrow t_1] \ldots [X_n \rightarrow t_n]$ is said to be of type $\text{Subs}(s)$ if

$$
\bigcup_{i=1}^{n} \text{Vars}((X_i, t_i)) \subseteq \text{Vars}(s) \text{ and } X_i \notin \text{Vars}(t_i), \forall i = 1, \ldots, n.
$$

**Lemma 4.7** (Decrement of variables for substitutions of type $\text{Subs}(s)$)

Let $\sigma$ be a substitution of type $\text{Subs}(s)$.
(i) $\text{Vars}(t\sigma) \subseteq \text{Vars}((t,s))$.

(ii) $\sigma \neq \text{Id}$ implies that $|\text{Vars}(t\sigma)| < |\text{Vars}((t,s))|$.

**Proof.** By induction on the length of $\sigma$.

(i) If $\sigma = \text{Id}$, then obviously $\text{Vars}(t) \subseteq \text{Vars}((t,s))$. If $\sigma = \sigma'[X \mapsto u]$, then $t\sigma = (t\sigma')[X \mapsto u]$. By induction hypothesis, $\text{Vars}(t\sigma') \subseteq \text{Vars}((t,s))$. As $X \notin \text{Vars}(u)$, it is known that

$$\text{Vars}(t\sigma) = \text{Vars}(t\sigma'[X \mapsto u]) = \text{Vars}((t\sigma',u)) \setminus \{X\} \subseteq \text{Vars}((t,s)) \setminus \{X\} \subseteq \text{Vars}((t,s)).$$

(ii) From (i), $\text{Vars}(t\sigma'[X \mapsto u]) \subseteq \text{Vars}((t,s)) \setminus \{X\}$. Since $X \in \text{Vars}(s)$, the cardinality indeed decreases, i.e., $|\text{Vars}((t,s)) \setminus \{X\}| = |\text{Vars}((t,s))| - 1$.

**Lemma 4.8 (Termination of unify)** If $\text{unify}(t,s) = (\nabla,\sigma,b)$, then the substitution $\sigma$ is of type $\text{Subs}((t,s))$.

**Proof.** This is easily checked observing the nuclear substitutions generated in the cases of suspended variables. Note that, one condition to build $[X \mapsto \pi^{-1} \cdot u]$, for instance, is $X \notin \text{Vars}(u)$.

The last two lemmas ensure termination for the function $\text{unify}$.

**Notation:** It is said that $\Delta \vdash \sigma \approx_\alpha \gamma$ if, for any $Y$, $\Delta \vdash (t \cdot Y)\sigma \approx_\alpha (t \cdot Y)\gamma$.

An auxiliary lemma regarding the action of $\alpha$-equivalent substitutions over a term is necessary for the formalisation of the completeness of the unification algorithm and it is presented below.

**Lemma 4.9** $\Delta \vdash \sigma \approx_\alpha \gamma$ implies $\Delta \vdash t\sigma \approx_\alpha t\gamma$, for all term $t$.

**Proof.** By induction on the structure of $t$.

The next results are the most difficult part of the formalisation (fully available at [trs.cic.unb.br](http://trs.cic.unb.br)). Soundness and completeness formalisations follow the same inductive proof technique and the analysis of cases are also analogous. Thus, we focus only on completeness.

**Lemma 4.10 (Soundness)** Let $(\nabla,\sigma,b)$ be the solution for $\text{unify}(t,s)$. If $b = True$, then $(\nabla,\sigma)$ is a unifier of $t$ and $s$.

**Proof.** The proof is by induction on $\text{lex}(|\text{Vars}((t,s))|,\text{depth}(t))$.

The previous lemma alone is not enough in the sense that, if the algorithm returns always $False$, then no unifier is provided, even to unifiable terms. The next theorem guarantees that the algorithm actually gives a unifier whenever the terms are unifiable and that the answer is the most general unifier.

**Definition 4.11 (More general solutions)** Let $\nabla,\Delta$ be two contexts and $\gamma,\sigma$ two substitutions. Then $(\nabla,\gamma) \leq (\Delta,\sigma)$ if there exists $\theta$ such that

$$\Delta \vdash \nabla \theta \text{ and } \Delta \vdash \gamma \theta \approx_\alpha \sigma.$$
If \((\nabla, \gamma)\) is the least unifier for a unification problem according to \(\leq\), then it is a most general unifier (mgu).

**Theorem 4.12 (Completeness)** Let \((\nabla, \gamma, b)\) be the solution for unify\((t, s)\). If there exists any other solution \((\Delta, \sigma)\) for the unification problem, i.e., \(\Delta \vdash t \sigma \approx_a s \sigma\), then \(b = True\) and \((\nabla, \gamma) \leq (\Delta, \sigma)\).

**Proof.** The proof is by induction on \(\text{lex}(\mid Vars(t, s) \mid, \text{depth}(t))\). There are some cases to consider: either \(t\) or \(s\) are suspensions or both have the same structure, that is, \(t\) and \(s\) are units or abstractions, for instance. That is due to the \(\alpha\)-equivalence between \(ts\) and \(s\sigma\) and the fact that \(\sigma\) cannot change the structure of a term, unless acting over suspended variables. Below, we present the cases where \(s\) is a suspension, both are pairs, and both are abstractions; these are the most interesting cases.

1. \((t, \pi \cdot X)\) and \(X \notin Vars(t)\) : so \(\Delta \vdash t \sigma \approx_a (\pi \cdot X) \sigma \equiv \pi \cdot (X \sigma)\) by Lemma 2.9.

2. \(\cdot Y \neq X\) implies \(\Delta \vdash Y[X \mapsto \pi^{-1} \cdot t]\) \(\leq (\Delta, \sigma)\). By definition of \(\leq\), it is necessary to prove \(\theta\) such that \(\forall Y : \Delta \vdash Y[X \mapsto \pi^{-1} \cdot t] \theta \approx_a Y \sigma\). Instantiate it with \(\theta\).

3. \(Y = X\) : \(\Delta \vdash t \sigma \approx_a \pi \cdot (X \sigma)\) implies \(\Delta \vdash \pi^{-1} \cdot t \sigma \approx_a X \sigma\), by Lemma 3.6. As \(X[X \mapsto \pi^{-1} \cdot t] \sigma \equiv \pi^{-1} \cdot t \sigma\), the \(\alpha\)-equivalence is derivable.

4. \(((t_1, t_2), (s_1, s_2))\) : by hypothesis, \(\Delta \vdash t_1 \sigma \approx_a s_1 \sigma\) and \(\Delta \vdash t_2 \sigma \approx_a s_2 \sigma\).

   By IH, \(\text{unify}(t_1, s_1) = (\nabla_1, \gamma_1, True)\) and \((\nabla_1, \gamma_1) \leq (\Delta, \sigma)\), i.e.,

   \[
   \text{there exists } \theta \text{ such that } \Delta \vdash \nabla_1 \theta \text{ and } \Delta \vdash \gamma_1 \theta \approx_a \sigma.
   \]

   By Lemma 4.9, transitivity and symmetry, \(\Delta \vdash t_2 \gamma_1 \theta \approx_a s_2 \gamma_1 \theta\), that is, \((\Delta, \theta)\) is a unifier for \(t_2 \gamma_1\) and \(s_2 \gamma_1\).

   Using IH again, with \(\text{unify}(t_2 \gamma_1, s_2 \gamma_1) = (\nabla_2, \gamma_2, True)\), we obtain \(\Delta \vdash \nabla_2 \theta\) and \(\Delta \vdash \gamma_2 \theta \approx_a \theta\) for some \(\theta\).

   As \(\text{unify}((t_1, t_2), (s_1, s_2)) = (\nabla_1 \nabla_2 \gamma_2, \gamma_1 \gamma_2, b)\), all we need to prove is that \(\Delta \vdash \gamma_1 \gamma_2 \theta \approx_a \sigma\) and \(\Delta \vdash \nabla_1 \gamma_2 \theta\) (because \(\Delta \vdash \nabla_1 \theta\) follows by IH).

   By Lemma 4.9, for any variable \(Y\), it is possible to derive

   \[
   \Delta \vdash (id \cdot Y \gamma_1) \gamma_2 \theta \approx_a (id \cdot Y \gamma_1) \theta \approx_a id \cdot Y \sigma.
   \]

   So, by transitivity, \(\Delta \vdash \gamma_1 \gamma_2 \theta \approx_a \sigma\) holds.

   Finally, as \(\Delta \vdash \gamma_2 \theta \approx_a \theta\) and \(\Delta \vdash \nabla_1 \theta\), then \(\Delta \vdash \nabla_1 \gamma_2 \theta\).

5. \(([a][b], [b][s])\) : by premisse, \(\Delta \vdash t \sigma \approx_a (a b) \bullet (s \sigma)\) by Lemma 2.9. \((a b) \bullet (s \sigma)\) and \(\Delta \vdash a \# \hat{s} \sigma\).

   By IH, \(\text{unify}(a b \bullet \hat{s} \sigma) = (\nabla_1, \gamma, True)\) and \((\nabla_1, \gamma) \leq (\Delta, \sigma)\), i.e.,

   \[
   \text{there is } \theta \text{ such that } \Delta \vdash \nabla_1 \theta \text{ and } \Delta \vdash \gamma \theta \approx_a \sigma.
   \]

   By Lemma 3.5, \(\Delta \vdash a \# \hat{s} \sigma\) implies \(\Delta \vdash a \# \hat{s} \gamma \theta\). As \(\theta\) cannot eliminate any inconsistency in \(a \# \hat{s} \gamma\), then \(\Delta \vdash a \# \hat{s} \gamma\).

   By Lemma 3.13, as \((a \# \hat{s} \gamma)_{sol}\) is complete, so \((a \# \hat{s} \gamma)_{sol} = (\nabla_2, True)\).

   Thus, the algorithm computes \(\text{unify}(a [a], [b] [s]) = (\nabla_2 \nabla_1, \gamma, True)\). To show that \((\nabla_1 \nabla_2, \gamma) \leq (\Delta, \sigma)\), we only need to see that \(\Delta \vdash \nabla_2 \theta\). Finally, once \(\nabla_2 \theta = (a \# \hat{s} \gamma)_{}\) then the result follows.
5 Conclusions and future work

In this work, a nominal unification algorithm that only takes terms as parameters was presented. Unlike other approaches, which use transformation rules and take the corresponding freshness problems as part of the unification problem, here we have designed a function that can compute the freshness contexts separately. Our nominal unification algorithm is more straightforward and closer to the ones that implement first-order unification.

Additionally, we formalised transitivity for \(\approx_\alpha\) in a direct manner without using a weak intermediate relation as in [Urb10b]. Here, the proof was based on elementary lemmas about permutations, freshness and \(\alpha\)-equivalence; such lemmas are well-known in the context of nominal unification. In [Urb10b], the same auxiliary lemmas to demonstrate transitivity were proved, including some extra lemmas to deal with this weak-equivalence. We believe that the current formalisation of transitivity of \(\approx_\alpha\) is simpler in the sense that it only uses the essential notions and results. Symmetry of \(\approx_\alpha\) is also formalised independently from transitivity, diverging from [UPG04,Urb10b].

An important aspect to stress is that the style of proof formalised here could have been done in any other higher-order proof assistant. But PVS was used having in mind the objective of enriching the libraries for term rewriting systems, as mentioned in the introduction. Important features of PVS such as dependent types can be replaced by other mechanisms in Isabelle/HOL, for instance. For example, the substitution generated in the computation of \(\text{unify}(t,s)\) must be of type \(\text{Subs}_\text{unif}(t,s)\) (this is the PVS specification for the type \(\text{Subs}((t,s))\) in Definition 4.6) in order to prove termination. In Isabelle/HOL, this is overcome by defining substitutions in a slightly different way. PVS also allows to use type variables when defining a theory; those variables can be parameterised when such theory is imported by another one. In Isabelle/HOL, parameterising theories is not straightforward, but functions can be defined polymorphically, which provides different feasible solutions for the same kind of formalisation. Of course, a formalisation in Isabelle/HOL will bring out the possibility of a direct comparison regarding the previous formalisations of unification in [Urb10a], but it should be emphasised that the advantages of the current formalisation arise from the differences in the theoretical proofs.

Future work: Although nominal approaches have several advantages in the treatment of bound variables, there is still work to be done regarding the study of relevant computational properties. At a first glance, a subsequent study to be done is applying nominal unification for the construction of a nominal completion algorithm à la Knuth-Bendix as part of a PVS development for nominal rewriting. A completion algorithm for closed nominal rewriting systems is provided in [FR12].

Another possible application of this formalisation of the nominal unification algorithm is in the verification of nominal resolution approaches (as done, for instance, in the propositional case in [CM09]).
References


Strong Normalization through Intersection Types and Memory

Antonio Bucciarelli Delia Kesner

Univ Paris Diderot, Sorbonne Paris Cité, PPS, CNRS, Paris, France

Daniel Ventura

Univ. Federal de Goiás, Instituto de Informática, Goiânia, Brazil

Abstract

We characterize $\beta$-strongly normalizing $\lambda$-terms by means of a non-idempotent intersection type system. More precisely, we first define a memory calculus $K$ together with a non-idempotent intersection type system $K$, and we show that a $K$-term $t$ is typable in $K$ if and only if $t$ is $K$-strongly normalizing. We then show that $\beta$-strong normalization is equivalent to $K$-strong normalization. We conclude since $\lambda$-terms are strictly included in $K$-terms.

Keywords: Lambda-calculus, memory calculus, strong normalization, intersection types.

1 Introduction

It is well known that the $\beta$-strongly normalizing $\lambda$-terms can be characterized as those being typable in suitable intersection type systems. This result dates back to the late 1970s and early 1980s, when intersection types were invented to endow the pure lambda calculus with powerful type-assignment systems [2,11,26,24]. A survey of these results, out of the scope of this paper, can be found for instance in [32,3].

In more recent years, a revisitation of those early results has been driven by the introduction of resource aware semantics of $\lambda$-calculi [19,6,14,7] and of the corresponding non-idempotent intersection types assignment systems.

Just like their idempotent precursors, these type systems allow for a characterization of strong normalization [5,13] (as well as weak normalization and head normalization [14,8]), but they also grant a substantial improvement: proving that typable terms are strongly normalizing becomes much simpler. Let us provide a brief account of this improvement, by highlighting in the way the quantitative character of non-idempotent intersection types versus the qualitative flavor of the idempotent ones. The proof of the statement above, in the non-idempotent case, goes roughly as follows: given a typing derivation for a term $t$, and willing to prove that $t$ is...
strongly normalizing, take whatever \( \beta \)-reduct \( t' \) of \( t \). The subject reduction lemma, in this case, ensures not only that \( t' \) is typable but also that there exists a typing derivation for \( t' \) whose size is smaller than the one of the typing derivation for \( t \) we started from. Hence any \( \beta \)-reduction sequence starting from \( t \) is finite.

This shrinking of the size of type derivations along reduction sequences, in sharp contrast to what happens in the idempotent setting, is essentially due to the fact that a type derivation for a term of the shape \( (\lambda x.u)v \) may require as many sub-derivations for \( v \) as the number of occurrences of \( x \) in \( u \).

Let us provide a simple example involving the Church numeral \( n := \lambda y.\lambda x.y(...y(x))... \).

Why is the term \( s = \lambda x.t(...t(x))... \), \( t \) being an arbitrarily complex term, “simpler to type” than its \( \beta \)-expanded form \( nt \)? The point is that the typical non-idempotent intersection type \(^2\) that can be assigned to the Church numeral \( n \) is, in our notation, \([\sigma] \rightarrow \sigma, ..., [\sigma] \rightarrow \sigma \rightarrow [\sigma] \rightarrow \sigma \), the leftmost multiset containing \( n \) copies of \( [\sigma] \rightarrow \sigma \). Thus, in order to assign a type to \( nt \), \( n \) typing derivations assigning \( [\sigma] \rightarrow \sigma \) to \( t \) must be provided, exactly like in a type derivation for \( s \). At the same time, the outermost application \( nt \rightarrow \beta s \), so the typing derivation for \( s \) is smaller than that for \( nt \).

In the idempotent case, on the other hand, a type \(^3\) for \( n \) is an instance of \( \{\sigma\} \rightarrow \sigma \rightarrow \sigma \), and the type derivations for \( s \) may be hugely bigger than those for \( nt \), the former requiring \( n \) sub-derivations for \( t \), the latter just one. That’s why, for idempotent intersection type systems, the proof of the result above cannot be combinatorial, and is typically based on the reducibility argument \(^{29,16,23}\).

This shift of perspective goes beyond lowering the logical complexity of the proof: the quantitative information provided by type derivations in the non-idempotent setting unveils interesting relations between typings (static) and reductions (dynamic) of \( \lambda \)-terms. For instance, in \(^{14}\), a correspondence between the size of a typing derivation for \( t \) and the number of steps taken by a Krivine machine to reduce \( t \) is presented, and in \(^{5}\) it is shown how to compute the length of the longest \( \beta \)-reduction sequence starting from any typable strongly normalizing \( \lambda \)-term.

In this paper, we provide a characterization of strongly normalizing \( \lambda \)-terms via a typing system based on non-idempotent intersection types. The structure of the proof is the following:

- We define the \( K \)-calculus, reminiscent of Klop’s \( I \)-calculus \(^{22}\), where terms are defined by enriching \( \lambda \)-terms with a memory operator, \( \beta \)-reduction is split into two different non-erasing reductions, and terms are considered modulo an equivalence relation, reminiscent of Regnier’s \( \sigma \)-reduction \(^{27}\). In contrast to \(^{22}\), \( \lambda \)-terms are strictly included in \( K \)-terms, which makes our development much easier.

- We introduce the typing system \( K \) for \( K \)-terms, based on system \( Q \) for focused intuitionistic logic \(^{17}\), and we show that a \( K \)-term is \( K \)-typable if and only if it is \( K \)-strongly normalizable. This proof is only based on typing properties of Subject Reduction and Subject Expansion, and does not use any reducibility argument.

---

1. More precisely, it requires exactly as many sub-derivations for \( v \) as the number of typed occurrences of \( x \) in \( u \).
2. Non-idempotent intersections are denoted by multisets, e.g. \([\tau, \tau, \sigma]\) stands for \( \tau \land \tau \land \sigma \).
3. Idempotent intersections are denoted by sets, e.g. \( \{\tau, \sigma\} \) stands for \( \tau \land \sigma \).
• We prove that λ-terms are K-strongly normalizable if and only if they are β-
strongly normalizable in the λ-calculus.

Related works: Several characterizations of strong normalization via idempotent
intersection types have been presented for the λ-calculus; a survey can be found
for example in [3]. To the best of our knowledge, two characterizations of strong
normalization via non-idempotent intersection types have been presented so far for
the λ-calculus, by A. Bernadet and S. Lengrand [5] and by E. De Benedetti and S.
Ronchi Della Rocca [13,12], respectively. Non-idempotent intersection is also used
in the systems of [20,15], both for the λ-calculus, but characterization of strong
normalization is achieved through a relation to an idempotent intersection type
system.

In [5], a subtyping relation is used to get the subject reduction property, but the
system types unnecessary instances of arguments, and turns out to be non relevant.
Moreover, notions of optimality and principality of typing derivations are necessary
to derive an exact upper bound for reduction steps, while in the present work, the
size of typing derivation trees is enough to measure them.

Besides, the “strong normalizing implies typability” property is obtained in [5]
through a subject expansion property on a restricted version of the β-reduction,
where a memory set is used to trace the free variables of erased terms. In our
memory calculus, which is non erasing, both the subject reduction and the subject
expansion properties hold unconditionally.

In [13] the typing rule for term variables is weakened, thus the system is non
relevant. The characterization of strong normalization, more precisely the “strong
normalization implies typability” property, is reported as following from an adap-
tation of the perpetuality proof in [25]. In [12] another proof of the same property
is obtained through an inductive definition of the set of strongly normalizing terms,
as done in [18] and in the present work.

In the extended framework of the λ-calculus with explicit substitution, non-
idempotent types were also used to characterize strong normalization [5,18]. In [5]
the typing system deals with two explicit substitution calculi based on the struc-
tural propagation paradigm, while in [18] the substitution at a distance paradigm is
investigated. In all the cases, the normalization property is proved by relying on
the postponement of erasing steps, where the explicit substitution operator plays
the role of a memory device. No exact bound for the normalization process of ex-
plicit substitution calculi is studied in those papers, even though some measures are
proposed in [5] for one of the non-erasing calculi.

Regarding the characterization of strong normalization for the sequent calculus,
[21] defines an idempotent IT system, based on a first proof given in [30]. This is
in the spirit of the proof for idempotent types presented in [3] (see Sec. 5).

Structure of the paper: Sec. 2 presents the syntax and semantics of the K-calculus,
while Sec. 3 introduces the non-idempotent typing system K for K-terms together
with its properties. The characterization of β and K-strongly normalizing terms is
developed in Sec. 4. For the sake of completeness, we recall in Sec. 5 an alternative

\footnote{The notion of principal typing in [5] is different from the usual definition in the literature, e.g. [28,34,20].}
characterization of $\beta$ strongly normalizing terms, which can easily be rephrased in our framework. We conclude in Sec. 6.

2 The memory calculus

We are going to characterize the set of strongly normalizing $\lambda$-terms by using a memory calculus called $K$-calculus – reminiscent of Klop’s $I$-calculus [22] – as main technical tool. This section introduces the syntax and the operational semantics of the $K$-calculus.

Given a countable infinite set of symbols $x, y, z, \ldots$ we define the set of $K$-terms by means of the following grammar.

$$t, u, v ::= x | \lambda x.t | tu | t[u]$$

The syntactic item $[u]$ is called a memory operator. Notice that the set of $\lambda$-terms is strictly included in the set of $K$-terms.

The size of a term $t$ is written $|t|$. The notions of free and bound variables are defined as usual, in particular, $fv(t[u]) := fv(t) \cup fv(u)$, $bv(t[u]) := bv(t) \cup bv(u)$. We work with the standard notion of $\alpha$-conversion i.e. renaming of bound variables. Substitutions are (finite) functions from variables to terms. We use the notation $\{x_1/u_1, \ldots, x_n/u_n\}$ ($n \geq 0$) for a finite substitution $\sigma$ such that $\sigma(x_i) = u_i$ for $1 \leq i \leq n$. Application of the substitution $\sigma$ to the term $t$, written $t\sigma$, may require $\alpha$-conversions in order to avoid the capture of free variables. Hence we follow the common practice of considering terms up to $\alpha$-equivalence. However, we feel free to represent $\alpha$-equivalence classes by any of their members, provided they respect the usual Barendregt’s convention [1] stipulating that the sets of free and bound variables of any term are disjoint.

The standard notion of $\beta$-reduction on $\lambda$-terms which is generated by the closure by contexts of the rewriting rule $(\lambda x.t)u \rightarrow_{\beta} t\{x/u\}$.

We now consider the following equation and rewriting rules on $K$-terms.

**Equation:**

$$t[u]v =_{\sigma} (tv)[u]$$

**Rules:**

$$(\lambda x.t)u \rightarrow_{neb} t\{x/u\} \text{ if } x \in fv(t)$$

$$(\lambda x.t)u \rightarrow_{m} t[u] \text{ if } x \notin fv(t)$$

The names $\text{neb}$ and $m$ mean, respectively, non-erasing beta and memory. The reduction relation $\rightarrow_{\text{neb,m}}$ is generated by the closure by contexts of the rewriting rules $\rightarrow_{\text{neb}}$ and $\rightarrow_{m}$. The relation $\sim_{\sigma}$ is the equivalence relation on $K$-terms generated by the equation $=_{\sigma}$. The $K$-calculus is given by the set of $K$-terms and the reduction relation $\rightarrow_{K}$ on $K$-terms, generated by the reduction $\rightarrow_{\text{neb,m}}$ modulo the equivalence $\sim_{\sigma}$. Thus for example $(\lambda x.\lambda y.z)x'y' \rightarrow_{m} (\lambda y.z)[x'][y'] \rightarrow_{n} z[y'][x']$, which is not K-reducible anymore, i.e. it is a $K$-normal form. Another examples is given by the following $K$-reduction inside a memory operator $y((\lambda w. (\lambda x.z)[z']x') \rightarrow_{\text{neb}} y[(\lambda w.z)[z']x'] \rightarrow_{m} y[z[x']][z'])$. 

38
Given a reduction relation $R$, a term $t$ is said to be $R$-strongly normalizing, written $t \in SN(R)$, iff there is no infinite $R$-reduction sequence starting at $t$.

3 The Typing System

In this section we introduce a type system for $K$-terms, called $\mathcal{K}$, whose intersection types, IT for short, are similar to those in [9,10].

Let $\mathcal{A}$ be a countable infinite set of type variables $\alpha, \beta, \gamma, \ldots$. The sets $\mathcal{T}$ of strict types, ranged over by $\sigma, \tau, \ldots$, and $\mathcal{U}$ of multiset types, ranged over by $\mathcal{M}, \mathcal{M}', \ldots$, are defined by the following grammars:

$$\sigma ::= \alpha | \mathcal{M} \rightarrow \sigma$$

$$\mathcal{M} ::= [\sigma_i]_{i \in I}, \text{ } I \text{ finite set}$$

Types are strict, i.e. multiset types do not occur on the right-hand sides of arrows. A multiset type should be read as the intersection of the strict types it contains. For instance, the multiset $[\tau, \tau, \sigma]$ stands for $\tau \land \tau \land \sigma$, where the symbol $\land$ is associative, commutative and non-idempotent. Observe however that the commutativity, associativity and non-idempotency of the intersection symbol is granted by the multiset notation: no further equivalence relation on types is needed.

The empty multiset is denoted by $[]$, while $+$ denotes multiset union.

Type assignments, written $\Gamma, \Delta$, are functions from variables to multiset types, assigning the empty multiset to all but a finite set of variables. The domain of $\Gamma$ is given by $\text{dom}(\Gamma) := \{x \mid \Gamma(x) \neq []\}$. The intersection of type assignments, written $\Gamma + \Delta$, is defined by $(\Gamma + \Delta)(x) := \Gamma(x) + \Delta(x)$. Hence, $\text{dom}(\Gamma + \Delta) = \text{dom}(\Gamma) \cup \text{dom}(\Delta)$. When $\text{dom}(\Gamma)$ and $\text{dom}(\Delta)$ are disjoint we write $\Gamma; \Delta$ instead of $\Gamma + \Delta$. We write $\Gamma \setminus x$ for the assignment $(\Gamma \setminus x)(x) = []$ and $(\Gamma \setminus x)(y) = \Gamma(y)$ if $y \neq x$.

The type assignment system $\mathcal{K}$ for $K$-terms is defined in Fig. 1. Notice that, in contrast to [4,13] the system is syntax directed.

![Fig. 1. The intersection type system for the $K$-calculus](image)

The rules $(\rightarrow_\epsilon)$ could be specified by means of two different typing rules separating the cases $I = \emptyset$ and $I \neq \emptyset$:

$$\Gamma \vdash t|I| \rightarrow \tau \quad \Delta \vdash u:\sigma \quad \Gamma + \Delta \vdash tu:\tau$$

$$\Gamma \vdash t|I| \rightarrow \tau \quad \Delta \vdash u:\sigma \quad \Gamma + \Delta \vdash tu:\tau$$

5 The name strict was coined by S. van Bakel in [31]

6 Differently from [10] that explicitly defines idempotency although it has the same notation for IT.
Indeed, if \( I = \emptyset \) in rule \((\rightarrow_e)\), then the argument \( u \) is erasable, so that we require exactly one typing derivation for \( u \), by setting \(|J| = 1\) (notice that the type \( \sigma \) is ignored in the final conclusion of the rule); otherwise the argument \( u \) is not erasable and several typing derivations for \( u \) are required, one for each type of the multiset \([\sigma_i]_{i \in I}\), thus \( J = I \). We prefer however to specify both cases in the single rule \((\rightarrow_e)\) in order to save some space in our proofs.

As an example of typing derivation in \( \mathcal{K} \), for \( t = \lambda x.xx \) one has

\[
\begin{align*}
&\frac{x: [\alpha] \to \beta \vdash x: [\alpha] \to \beta}{x: [\alpha], [\alpha] \to \beta \vdash xx: \beta} \\
&\vdash \lambda x.xx: [\alpha], [\alpha] \to \beta \to \beta
\end{align*}
\]

hence

\[
\begin{align*}
&\frac{z: [\gamma] \vdash z: \gamma}{z: [\gamma] \vdash \lambda y.z: [] \to \gamma} \quad \vdash t: [\alpha], [\alpha] \to \beta \to \beta \\
&\vdash z: [\gamma] \vdash (\lambda y.z): t: \gamma
\end{align*}
\]

and also \( z: [\gamma] \vdash z[t]: \gamma \). On the other hand, neither \((\lambda y.z): \Omega\) nor \( z[\Omega] \) are typable in \( \mathcal{K} \) for \( \Omega = tt \).

We write \( \Phi \vdash \Gamma \vdash t: \tau \) to denote typing derivability in system \( \mathcal{K} \). For any typing derivation tree \( \Phi \), we define \( sz(\Phi) \) to be the number of nodes of \( \Phi \).

The \( \mathcal{K} \)-system enjoys relevance (absence of weakening).

**Lemma 3.1 (Relevance)** If \( \Phi \vdash \Gamma \vdash t: \tau \) then \( dom(\Gamma) = fv(t) \).

**Proof.** By induction on the derivation of \( \Phi \).

Moreover, the equivalence relation \( \sim_\sigma \) does not alter at all the typing relation.

**Lemma 3.2 (Typing Invariance for \( \sim_\sigma \))** Let \( t_0 \sim_\sigma t'_0 \). Then \( \Phi \vdash \Gamma \vdash t_0: \tau \) iff \( \Phi' \vdash \Gamma \vdash t'_0: \tau \). Moreover, \( sz(\Phi) = sz(\Phi') \).

**Proof.** By induction on the proof of \( t_0 \sim_\sigma t'_0 \). We only show the base case, the others being straightforward.

If \( t_0 = t[u]:v =_\sigma (tv)[u] = t'_0 \), then by construction \( \Gamma = \Delta + \Pi + j \in J \; \Gamma_j \) and \( \Phi \) is of the following form:

\[
\begin{align*}
\frac{\Phi_t \triangleright \Delta \vdash t: [\sigma_i]_{i \in I} \to \tau \quad \Phi_u \triangleright \Pi \vdash u: \rho}{\Delta + \Pi \vdash t[u]: [\sigma_i]_{i \in I} \to \tau} \quad \frac{\Phi'_t \triangleright \Gamma_j \vdash v: \sigma_j, j \in J}{\Delta + \Pi + j \in J \; \Gamma_j \vdash t[u]: \tau} \quad \frac{\Phi_u \triangleright \Pi \vdash u: \rho}{\Delta + j \in J \; \Gamma_j + \Pi = (tv)[u]: \tau}
\end{align*}
\]

where \(|J| = 1\) if \( I = \emptyset \) and \( I = J \) otherwise. Moreover, \( sz(\Phi) = sz(\Phi_t) + sz(\Phi_u) + j \in J \; sz(\Phi'_t) + 2 \). Then,

\[
\Phi' := \frac{\Phi_t \triangleright \Delta \vdash t: [\sigma_i]_{i \in I} \to \tau \quad \Phi'_t \triangleright \Gamma_j \vdash v: \sigma_j, j \in J}{\Delta + j \in J \; \Gamma_j + \Pi = (tv)[u]: \tau} \quad \frac{\Phi_u \triangleright \Pi \vdash u: \rho}{\Delta + \Pi + j \in J \; \Gamma_j \vdash t[u]: \tau}
\]
where \( \text{sz}(\Phi') = \text{sz}(\Phi_i) + \sum_{j \in I} \text{sz}(\Phi_j) + \text{sz}(\Phi_u) + 2 = \text{sz}(\Phi) \).

We are now going to show the essential parts of the typing system \( \mathcal{K} \):
Subject Reduction (Thm. 3.4) and Subject Expansion (Thm. 3.6), which follow, respectively, from Lem. 3.3 and Lem. 3.5.

**Lemma 3.3 (Substitution Lemma)** If \( \Phi \triangleright x ; [\rho] ; i \in I ; \Gamma \vdash t : \tau \) and \( (\Phi' \triangleright \Delta_i \vdash u ; \rho_i)_{i \in I} \) then \( \Phi_{t(x/u)} \triangleright \Gamma + i \in I \Delta_i \vdash t[x/u] : \tau \) where \( \text{sz}(\Phi_{t(x/u)}) = \text{sz}(\Phi_i) + i \in I \text{sz}(\Phi_u) - |I| \).

**Proof.** By induction on the structure of \( \tau \).

- If \( t = y \neq x \) then \( t[x/u] = y \). By construction one has \( x : [\tau] ; \Gamma = \{ y : [\tau] \} \) so that \( I = \emptyset \) and \( \Gamma = \{ y : [\tau] \} \) and \( \text{sz}(\Phi) = 1 \). Therefore, \( \text{sz}(\Phi_y) = \text{sz}(\Phi_i) + 0 - 0 = \text{sz}(\Phi_i) + i \in I \text{sz}(\Phi_u) - |I| \).

- If \( t = x \) then \( t[x/u] = u \). By construction one has \( x : [\rho] ; i \in I ; \Gamma = \{ x : [\tau] \} \) so that \( I = \{ m \} \) and \( \Gamma = \emptyset \) and \( \rho_m = \tau \) and \( \text{sz}(\Phi) = 1 \). Therefore, for any context \( \Delta_m \) such that \( \Phi_u \triangleright \Delta_m \vdash u ; \rho_m \) the result holds, where \( \text{sz}(\Phi_{x(x/u)}) = \text{sz}(\Phi_x) + i \in I \text{sz}(\Phi_u) - 1 = \text{sz}(\Phi_u) - 1 \).

- If \( t = \lambda y . v \) then by \( \alpha \)-conversion one can suppose w.l.o.g. that \( y \neq x \) and \( y \notin \text{dom}(\Gamma) \) and \( (y \notin \text{dom}(\Delta_i))_{i \in I} \). By construction, \( \tau = M \rightarrow \sigma \) and \( \Phi \triangleright y ; M ; x : [\rho] ; i \in I ; \Gamma \vdash v \cdot \sigma \) where \( \text{sz}(\Phi_v) + 1 = \text{sz}(\Phi_{\lambda y . v}) \).

The derivation of \( \Phi_{\lambda y . v (x/u)} \) is hence

\[
\frac{\Phi_{v(x/u)} \triangleright y ; M ; \Gamma + i \in I \Delta_i \vdash v \cdot x/u \cdot \sigma}{\Gamma + i \in I \Delta_i \vdash \lambda y . v \cdot x/u ; M \rightarrow \sigma} (\gamma_i)
\]

and \( \text{sz}(\Phi_{\lambda y . v (x/u)}) = \text{sz}(\Phi_{v(x/u)}) + 1 = \text{sz}(\Phi_{\lambda y . v}) + i \in I \text{sz}(\Phi_u) - |I| \).

- If \( t = p v \) then by construction one has \( x : [\rho] ; i \in I ; \Gamma = \Delta + i \in J \Gamma_j \), where \( \Phi \triangleright \Delta \vdash p ; [\sigma] ; k \in K \rightarrow \tau \) and \( (\Phi_j \triangleright \Delta_i \vdash v \cdot \sigma)_{j \in J} \) and \( \text{sz}(\Phi_{pv}) = \text{sz}(\Phi_p) + j \in J \text{sz}(\Phi_{\tau}) + 1 \), where either \( K = \emptyset \) and \( |J| = 1 \), or \( K = J \). One has \( \Delta = x : [\rho] ; i \in I \Delta_i \) and \( \Gamma_j = x : [\rho] ; i \in I \Gamma_j ; j \in J \) where \( I = I_p \cup \cup J_j I_j \) and \( I_p , (I_j) \in J \) can be assumed to be pairwise disjoint sets w.l.o.g. By the i.h., \( \Phi_{p(x/u)} \triangleright \Delta' + i \in I_{\Delta_i} \Delta_i \vdash p \cdot x/u \cdot [\sigma] ; k \in K \rightarrow \tau \) and \( (\Phi_{v(x/u)} \triangleright \Gamma_j + i \in I_{\Delta_j} \Delta_j \vdash v \cdot x/u \cdot \sigma)_{j \in J} \) where \( \text{sz}(\Phi_{pv(x/u)}) =\text{sz}(\Phi_p) + j \in I_p \text{sz}(\Phi_u) - |I_p| \) and \( \text{sz}(\Phi_{\Delta'}) = \text{sz}(\Phi_{\Delta}) + i \in I_{\Delta_i} \text{sz}(\Phi_u) - |I_j| \), for each \( j \in J \). Note that \( \Delta' + j \in J \Gamma_j = x : [\rho] ; i \in I ; \Gamma \parallel x = \Gamma \) and that \( + i \in I_{\Delta_i} \Delta_i + j \in J \Delta_j = \Delta + i \in J \Delta_j \). Therefore,

\[
\Phi_{(pv)(x/u)} : = \frac{\Phi_{p(x/u)} \triangleright \Delta' + i \in I_{\Delta_i} \Delta_i \vdash p \cdot x/u \cdot [\sigma] ; k \in K \rightarrow \tau}{\Gamma + i \in I \Delta_i \vdash (pv) \cdot x/u \cdot \tau} (\gamma_{ij})_{j \in J}
\]

where \( \text{sz}(\Phi_{(pv)(x/u)}) = \text{sz}(\Phi_{p(x/u)}) + j \in J \text{sz}(\Phi_{v(x/u)}) + 1 = \text{sz}(\Phi_p) + j \in I_{\Delta_i} \text{sz}(\Phi_u) - |I_p| + \sum_{j \in J} \text{sz}(\Phi_{\Delta_j}) + j \in J \text{sz}(\Phi_{\Delta_j}) - j \in J |I_j| + 1 = \text{sz}(\Phi_{pv}) + i \in I \text{sz}(\Phi_u) - |I| \).

- If \( t = p [v] \) then by construction \( x : [\rho] ; i \in I ; \Gamma = \Delta + \Pi + \Sigma \) such that \( \Phi \triangleright \Delta \vdash p \cdot \tau \) and \( \Phi \triangleright \Pi \vdash v \cdot \sigma \), where \( \text{sz}(\Phi_i) = \text{sz}(\Phi_p) + \text{sz}(\Phi_v) + 1 \). One has \( \Delta = x : [\rho] ; i \in I_p ; \Delta' \) and \( \Pi = x : [\rho] ; i \in I_v ; \Pi' \) where \( I_p \cup I_v = I \) and \( \Delta' + \Pi' = \Gamma \). Suppose w.l.o.g. that
$I_p$ and $I_v$ are disjoint. By the i.h., $\Phi_{p[x/u]} \triangleright \Delta' + i \in I_p \Delta_i \vdash p[x/u]:\tau$ and $\Phi_{v[x/u]} \triangleright \Pi' + i \in I_v \Delta_i \vdash v[x/u]:\sigma$ such that $sz(\Phi_{p[x/u]}) = sz(\Phi_p) + i \in I_p sz(\Delta_i) - |I_p|$ and $sz(\Phi_{v[x/u]}) = sz(\Phi_v) + i \in I_v sz(\Delta_i) - |I_v|$. Then,

$$\Phi_{p[x/u]} := \Phi_{p[x/u]} \triangleright \Delta' + i \in I_p \Delta_i \vdash p[x/u]:\tau \quad \Phi_{v[x/u]} \triangleright \Pi' + i \in I_v \Delta_i \vdash v[x/u]:\sigma$$

$$\Gamma + i \in I \Delta_i \vdash p[x/u][v[x/u]_i]:\tau$$

where $sz(\Phi_{p[x/u]}) = sz(\Phi_p) + sz(\Phi_{v[x/u]}) + 1 = sz(\Phi_t) + i \in I sz(\Delta_i) - |I|$. \hfill \qed

**Theorem 3.4 (Weighted Subject Reduction)** Let $\Phi \triangleright \Gamma \vdash t:\tau$. If $t \xrightarrow{K} t'$ then $\Phi' \triangleright \Gamma \vdash t':\tau$ such that $sz(\Phi) > sz(\Phi')$.

**Proof.** By induction on the reduction relation $\xrightarrow{K}$ using Lem. 3.2 to justify the statement for the equivalence relation $\sim_{\sigma}$.

- If $t = (\lambda x.v)u$ then by construction $\Gamma = \Delta + j \in J \Gamma_j$ and $\Phi$ has the form:

$$\Phi := \Delta \vdash \lambda x.v:[\sigma_i]_{i \in I} \rightarrow \tau \quad \Phi' \vdash \Gamma_j \vdash u:\sigma_j_{j \in J}$$

Moreover, $sz(\Phi) = sz(\Phi_v) + j \in J sz(\Delta_i) + 2$. There are two cases for $t'$.

- If $x \notin fv(v)$ then $t' = v[x/u]$. Suppose $I = \emptyset$ so that $\Gamma = \Delta + \Gamma_u$ and $\Phi \triangleright \Delta \vdash v:\tau$ where $x \notin dom(\Delta)$. However, $x \in dom(\Phi_v)$, then, by Lem. 3.1, this case is possible.

- Suppose $I \neq \emptyset$ (i.e., $I = J$) so that $\Gamma = \Delta + i \in I \Gamma_i$ where $\Phi \triangleright x:[\sigma_i]_{i \in I} \vdash v:\tau$ and $\Phi' \triangleright \Gamma_i \vdash u:\sigma_{j \in J}$ and $sz(\Phi) = sz(\Phi_v) + i \in I sz(\Delta_i) + 2$. By Lem. 3.3, $\Phi_{v[x/u]} \triangleright \Gamma \vdash v[x/u]:\tau$ where $sz(\Phi_{v[x/u]}) = sz(\Phi_v) + i \in I sz(\Delta_i) - |I| < sz(\Phi)$. We are done with this case taking $\Phi' := \Phi_{v[x/u]}$.

- If $x \notin fv(v)$ then $t' = v[u]$. Moreover, Lem. 3.1 gives $I = \emptyset$ so that $J = \{j\}$ where $\Gamma = \Delta + \Gamma_j$. Then,

$$\Phi' := \Phi \triangleright \Delta \vdash v:\tau \quad \Phi' \vdash \Gamma_j \vdash u:\sigma_j$$

where $sz(\Phi') = sz(\Phi_v) + sz(\Delta_i) + 1 < sz(\Phi)$. 

- All the inductive cases are straightforward. \hfill \qed

To obtain Subject Expansion we first need the following property.

**Lemma 3.5 (Reverse Substitution)** Let $x, t, u \in \Lambda_{K}$. If $x \notin fv(t)$ and $\Phi \triangleright \Gamma \vdash t[x/u]:\tau$ then $\Gamma = \Delta + i \in I \Gamma_i$ where $I \neq \emptyset$ and $\Phi_i \triangleright x:[\sigma_i]_{i \in I} \vdash \Delta \vdash t:\tau$ and $(\Phi_u \triangleright \Gamma_i \vdash u:\sigma_i)_{i \in I}$.

**Proof.** By induction on the structure of $t$. 

42
If \( t = x \) then \( \{x/u\} = u \) and the result holds, for \( \Delta = \emptyset \) and \( I = \{m\} \) where \( \sigma_m = \tau \). The derivation \( \Phi_i \) is obtained by an application of rule (ax). Observe that \( t = y \) would imply \( y \notin \mathcal{F}(t) \).

If \( t = \lambda y.v \) then \( \{x/u\} = \lambda y.x \{x/u\} \) thus, by construction, \( \tau = \mathcal{M} \rightarrow \sigma \) where \( \Phi \triangleright y.M; \Gamma \vdash v\{x/u\}:\sigma \). By the i.h., \( y.M; \Gamma = \Delta' + i \in I \Gamma_i \) where \( \Phi_u \triangleright x: [\sigma]_i \in I; \Delta' \vdash v\sigma \) and \( (\Phi_u \triangleright \Gamma_i \vdash u: [\sigma]_i)_i \in I \). Since \( y \notin \mathcal{F}(u) \) by Lem. 3.1 one has \( \Delta' = y.M; x: [\sigma]_i \in I; \Delta \) and \( (x: [\sigma]_i \in I; \Delta) + i \in I \Gamma_i = \Gamma \). Then \( \Phi_u \triangleright x: [\sigma]_i \in I; \Delta \vdash \lambda y.v: \tau \) by rule \((\rightarrow_i)\).

If \( t = uv \) then \( \{x/u\} = p\{x/u\} \{v\} \) and by construction \( \Gamma = \Pi + j \in IJ \Pi_j \) and \( \Phi_{p(x/u)} \triangleright \Pi \vdash p\{x/u\}: [\rho]_k \in K \rightarrow \tau \) and \( (\Phi_{x/u} \triangleright \Pi_j \vdash v\{x/u\}: [\rho]_j)_j \in J \) where either \( K = \emptyset \) and \( |J| = 1 \) or \( K = J \).

If either \( x \notin \mathcal{F}(v) \) or \( x \notin \mathcal{F}(p) \) then it is analogous to the case above.

If \( t = p'v \) then \( \{x/u\} = p\{x/u\}[v\{x/u\}] \) and by construction \( \Gamma = \Pi_0 + \Pi_1 \) such that \( \Phi_{p(x/u)} \triangleright \Pi_0 \vdash p\{x/u\}: \tau \) and \( \Phi_{x/u} \triangleright \Pi_1 \vdash v\{x/u\}: \sigma \).

If \( x \notin \mathcal{F}(p) \) and \( x \notin \mathcal{F}(v) \) then by i.h. \( \Pi_0 = \Delta_p + i \in I_P \Gamma_i \) where \( \Phi_p \triangleright x: [\sigma]_i \in I_P; \Delta_p \vdash p_i: \tau \) and \( (\Phi_p \triangleright \Gamma_i \vdash u: [\sigma]_i)_i \in I_P \) and \( \Pi_1 = \Delta_u + i \in I_u \Gamma_i \) where \( \Phi_u \triangleright x: [\sigma]_i \in I_u; \Delta_u \vdash v: \sigma \) and \( (\Phi_u \triangleright \Gamma_i \vdash u: [\sigma]_i)_i \in I_u \). Suppose w.l.o.g. that \( I_p \) and \( I_u \) are disjoint and let \( I = I_p \cup I_u \). Then \( \Phi_i \triangleright x: [\sigma]_i \in I; \Delta_p + \Delta_u \vdash p\{v\}: \tau \) by the rule \((m)\) and we are done with \( \Delta := \Delta_p + \Delta_u \).

If \( x \notin \mathcal{F}(v) \) then \( v\{x/u\} = v \) and by the i.h. one has \( \Pi_0 = \Delta_p + i \in I \Gamma_i \) such that \( \Phi_p \triangleright x: [\sigma]_i \in I; \Delta_p \vdash p\tau \) and \( (\Phi_u \triangleright \Gamma_i \vdash u: [\sigma]_i)_i \in I \). Then, by rule \((m)\), \( \Phi_i \triangleright x: [\sigma]_i \in I; \Delta_p + \Pi_1 \vdash \{p\}v: \tau \) and we are done with \( \Delta := \Delta_p + \Pi_1 \).

If \( x \notin \mathcal{F}(p) \) then it is analogous to the one above.

**Theorem 3.6 (Subject Expansion)** Let \( \Phi' \triangleright \Gamma \vdash t: \tau \). If \( t \rightarrow_k t' \) then \( \Phi' \triangleright \Gamma \vdash t': \tau \).

**Proof.** The proof is by induction on the reduction relation \( \rightarrow_k \).

- If \( t = (\lambda x.u) \rightarrow_{\text{neb}} v\{x/u\} = t' \), where \( x \notin \mathcal{F}(v) \), then by Lem. 3.5 \( \Delta = \Delta + i \in I \Gamma_i \) where \( I \neq \emptyset \), \( \Phi_u \triangleright x: [\sigma]_i \in I; \Delta \vdash v: \tau \) and \( (\Phi_u \triangleright \Gamma_i \vdash u: [\sigma]_i)_i \in I \). Then

  \[
  \Phi_u \triangleright x: [\sigma]_i \in I; \Delta \vdash v: \tau
  \]

  \[
  \Delta \vdash \lambda x.u: [\sigma]_i \in I \rightarrow \tau
  \]

  \[
  \Phi_u \triangleright \Gamma_i \vdash u: [\sigma]_i \in I
  \]

  \[
  \rightarrow_e
  \]

  \[
  \Gamma \vdash (\lambda x.u): \tau
  \]

  \[
  \rightarrow_e
  \]

  \[
  43
  \]
• If $t = (\lambda x.v)u \mapsto^m v[u] = t'$, where $x \notin \mathrm{fv}(v)$ then, by construction, $\Gamma = \Delta + \Gamma_1$ such that $\Phi_v : \Delta \vdash v : \tau$ and $\Phi_u : \Gamma_1 \vdash u : \sigma$. Note that, by Lem. 3.1, $x \notin \mathrm{dom}(\Delta)$. Then,

$$\frac{\Phi_v : \Delta \vdash v : \tau}{\Delta \vdash \lambda x.v : \tau \rightarrow \cdot} \quad \frac{\Phi_u : \Gamma_1 \vdash u : \sigma}{\Gamma \vdash (\lambda x.v)u : \tau}$$

• All the inductive cases are straightforward. □

4 The Strong Normalization Characterization

In this section we use $K$-typability to characterize $\beta$-strongly normalizing terms, i.e. we show that a term is $K$-typable if and only if it is $\beta$-strongly normalizing. The proof does not use any reducibility/computability argument [29,16,23], but goes through the memory operator calculus $K$ that we have introduced in Sec. 2. More precisely, we first show that $K$-terms are $K$-typable if and only if they are $K$-strongly normalizing. This proof is based on the properties presented in Sec. 3, namely, Weighted Subject Reduction and Subject Expansion. We then show that the sets of $\beta$ and $K$ strongly normalizing $\lambda$-terms are equivalent, a result which is obtained by means of appropriate inductive definitions for both sets.

It is well-known [33] that the set of $\beta$-strongly normalizing $\lambda$-terms can be characterized by the following inductive definition:

• If $t_1, \ldots, t_n$ ($n \geq 0$) $\in \mathcal{ISN}(\beta)$, then $xt_1 \ldots t_n \in \mathcal{ISN}(\beta)$.
• If $t \in \mathcal{ISN}(\beta)$, then $\lambda x.t \in \mathcal{ISN}(\beta)$.
• If $t[u]t_1 \ldots t_n, u \in \mathcal{ISN}(\beta)$, then $(\lambda x.t)u t_1 \ldots t_n \in \mathcal{ISN}(\beta)$.

This is indeed a characterization, as expressed by the following Lemma:

Lemma 4.1 ([33]) $\mathcal{SN}(\beta) = \mathcal{ISN}(\beta)$.

In the same spirit, we can give an inductive definition of the set of $K$-strongly normalizing $K$-terms:

(H) If $t_1, \ldots, t_n$ ($n \geq 0$) $\in \mathcal{ISN}(K)$, then $xt_1 \ldots t_n \in \mathcal{ISN}(K)$.
(A) If $t \in \mathcal{ISN}(K)$, then $\lambda x.t \in \mathcal{ISN}(K)$.
(I) If $t[x/u]t_1 \ldots t_n \in \mathcal{ISN}(K)$ and $x \notin \mathrm{fv}(t)$, then $(\lambda x.t)u t_1 \ldots t_n \in \mathcal{ISN}(K)$.
(G) If $t[u]t_1 \ldots t_n \in \mathcal{ISN}(K)$ and $n \geq 1$, then $t[u]t_1 \ldots t_n \in \mathcal{ISN}(K)$.
(S) If $(tt_1 \ldots t_n)[u] \in \mathcal{ISN}(K)$ and $n \geq 1$, then $t[u]t_1 \ldots t_n \in \mathcal{ISN}(K)$.
(V) If $t, u \in \mathcal{ISN}(K)$, then $t[u] \in \mathcal{ISN}(K)$.

The set $\mathcal{ISN}(K)$ turns out to be equivalent to the set of $K$-strongly normalizing terms. In order to show that, we write $\eta_K(t)$ to denote the maximal length of a $K$-reduction sequence starting at $t$, when $t$ is $K$-strongly normalizing.

Lemma 4.2 $\mathcal{SN}(K) = \mathcal{ISN}(K)$. 

Proof. If \( t \in SN(K) \), then we easily show that \( t \in ISN(K) \) by induction on the pair \( \langle \eta_K(t) \rangle \) using the corresponding lexicographic order. We only detail here the case \( t[u_1 \ldots t_n] \in SN(K) \), the other ones being easier. By definition we have 
\[
\eta_K(t[u_1 \ldots t_n]) = \eta_K((t_1 \ldots t_n)[u]),
\]
so that \( tt_1 \ldots t_n, u \in SN(K) \). By the i.h. we get 
\[
(t_1 \ldots t_n, u) \in ISN(K),
\]
by rule (V) we get \( (t_1 \ldots t_n)[u] \in ISN(K) \), and then by rule (S) we obtain \( t[u_1 \ldots t_n] \in ISN(K) \).

For the converse, we reason by induction on the definition of \( t \in ISN(K) \).

(H) If \( t = xt_1 \ldots t_n \in ISN(K) \), where \( t_1, \ldots, t_n \in ISN(K) \), then the i.h. gives 
\( t_1, \ldots, t_n \in SN(K) \) so that the term \( xt_1 \ldots t_n \) is trivially in \( SN(K) \).

(A) If \( t = \lambda x.v \in ISN(K) \), where \( v \in ISN(K) \), then the i.h. gives \( v \in SN(K) \) so that 
the term \( \lambda x.v \) is trivially in \( SN(K) \).

(I) If \( t = (\lambda x.v)ut_1 \ldots t_n \in ISN(K) \), where \( v[x/u]t_1 \ldots t_n \in ISN(K) \) and \( x \notin fv(v) \), 
then the i.h. gives \( v[x/u]t_1 \ldots t_n \in SN(K) \) so that in particular \( v, u, t_i \in SN(K) \). We show that 
\( t \in SN(K) \) by a second induction on \( \eta_K(v) + \eta_K(u) + \Sigma_{i=1 \ldots n}\eta_K(t_i) \).

Let us see how are all the reducts of \( t \).

If \( t \rightarrow (\lambda x.v')ut_1 \ldots t_n = t' \), where \( v \rightarrow v' \) or \( t \rightarrow (\lambda x.v)u't_1 \ldots t_n = t' \), where 
\( u \rightarrow u' \), or \( t \rightarrow (\lambda x.v)ut_1 \ldots t'_i \ldots t_n = t' \), where \( t_i \rightarrow t'_i \), then \( t' \in SN(K) \) by the second i.h.

If \( t \rightarrow v[x/u]t_1 \ldots t_n = t' \), then \( t' \in SN(K) \) as already remarked by the first i.h.

Since all reducts of \( t \) are in \( SN(K) \), then \( t \in SN(K) \).

(G) If \( t = (\lambda x.v)ut_1 \ldots t_n \in ISN(K) \), where \( v[u]t_1 \ldots t_n \in ISN(K) \) and \( x \notin fv(v) \), 
then the i.h. gives \( v[u]t_1 \ldots t_n \in SN(K) \), so that in particular \( v, u, t_i \in SN(K) \). We show that 
\( t \in SN(K) \) by induction on \( \eta_K(v) + \eta_K(u) + \Sigma_{i=1 \ldots n}\eta_K(t_i) \).

As before, let us analyze the reducts of \( t \).

If \( t \rightarrow (\lambda x.v')ut_1 \ldots t_n = t' \), where \( v \rightarrow v' \) or \( t \rightarrow (\lambda x.v)u't_1 \ldots t_n = t' \), where 
\( u \rightarrow u' \), or \( t \rightarrow (\lambda x.v)ut_1 \ldots t'_i \ldots t_n = t' \), where \( t_i \rightarrow t'_i \), then \( t' \in SN(K) \) by the second i.h.

If \( t \rightarrow v[u]t_1 \ldots t_n = t' \), then we have remarked already that \( t' \in SN(K) \) by the first i.h.

Since all reducts of \( t \) are in \( SN(K) \), then \( t \in SN(K) \).

(S) If \( v[u]t_1 \ldots t_n \in ISN(K) \) where \( (vt_1 \ldots t_n)[u] \in ISN(K) \) and \( n \geq 1 \), then the 
only possibility is \( vt_1 \ldots t_n, u \in ISN(K) \). We can then apply the i.h. to get 
\( vt_1 \ldots t_n, u \in SN(K) \), so that \( (vt_1 \ldots t_n)[u] = v[u]t_1 \ldots t_n \in SN(K) \).

(V) If \( t = v[u] \in ISN(K) \), where \( v, u \in ISN(K) \), then the i.h. gives \( u, v \in SN(K) \) so that the term \( v[u] \) is trivially in \( SN(K) \).

Since \( SN(K) \) and \( ISN(K) \) are equivalent sets, we can now derive \( K \)-typability from \( K \)-strong normalization by using this equivalence.

**Theorem 4.3** If \( t \in SN(K) \) then \( t \) is \( K \)-typable.

**Proof.** By induction on the structure of \( t \in ISN(K) = SN(K) \).

(H) If \( t = xt_1 \ldots t_n \) \((n \geq 0) \) where \( t_1, \ldots, t_n \in ISN(K) \) then, by the i.h. \( \forall 1 \leq i \leq n, \Gamma_i \vdash t_i : \sigma_i \). Let \( \tau = [\sigma_1] \rightarrow \cdots [\sigma_n] \rightarrow \alpha \), for \( \alpha \in A \), and \( \Gamma = x : [\tau] + \Gamma_1 + \cdots + \Gamma_n \).
Then, \( x: [\tau] \vdash x: \tau \) by the rule \((\text{ax})\) and, by \(n\) applications of the rule \((\rightarrow e)\), 
\( \Gamma \vdash x: \tau \) by the rule \((\text{ax})\) and, by \(n\) applications of the rule \((\rightarrow e)\), 
\( \Gamma \vdash_{t_1 \cdots t_n: \alpha} v t_1 \cdots t_n = t' \in SN(\beta) \) and thus we are done.  

**Lemma 4.6** Let \( t \) be a \( \lambda \)-term. If \( t \in SN(\beta) \), then \( t \in SN_K(\beta) \).

**Theorem 4.4** If \( t \) is \( K \)-typable then \( t \in SN(\beta) \).

**Proof.** Let \( \Phi \vdash \Gamma \vdash t: \tau \) and suppose that \( t \notin SN(\beta) \). Then, there exists an infinite reduction sequence \( t = t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow \cdots \rightarrow t_{2n+1} \rightarrow t_{2n+2} \rightarrow \cdots \). By Thm. 3.4 and Lem. 3.2 one has that \( \forall i \in \mathbb{N} \exists \Phi_i \vdash \Gamma \vdash t_i: \tau \) such that \( sz(\Phi_{2i}) > sz(\Phi_{2i+1}) \). A contradiction, since \( sz(\Phi) > 0 \), for any \( \Phi \). Therefore, \( t \in SN(\beta) \).

We have already remarked that the set of \( \lambda \)-terms is included in the set of \( K \)-terms. In order to conclude, we need to show that a \( \lambda \)-term \( t \) is \( \beta \)-strongly normalizable if and only if the \( K \)-term obtained by embedding \( t \) in the \( K \)-calculus is \( K \)-strongly normalizable.

**Lemma 4.5** Let \( t \) be a \( \lambda \)-term. If \( t \in SN(\beta) \), then \( t \in SN(\beta) \).

**Proof.** By induction on \( (\eta_\beta(t), [t]) \). We only show the interesting case. Let \( t = (\lambda x. v) u t_1 \cdots t_n \). By hypothesis \( v, u, t_i \in SN(\beta) \) so that the i.h. we get \( v, u, t_i \in SN(\beta) \). To prove that \( t \in SN(\beta) \) it is sufficient to show that every \( \beta \)-reduct of \( t \) is in \( SN(\beta) \). We proceed by induction on \( \eta_\beta(v) + \eta_\beta(u) + \sum_{i=1}^{n} \eta_\beta(t_i) \).

If \( t \rightarrow_\beta t' \) reduces a subterm \( v, u, t_i \), then the property trivially holds.

Otherwise, if \( t \rightarrow_\beta t' \) reduces the head redex, there are two cases to consider. If \( x \in fv(v) \), then also \( t \rightarrow_\beta v(x/u) t_1 \cdots t_n = t' \), so that \( t' \in SN(\beta) \) by the first i.h. If \( x \notin fv(v) \), then \( t \rightarrow_\beta v t_1 \cdots t_n = t' \). However, \( t \rightarrow_\beta v u t_1 \cdots t_n = t' \equiv (v t_1 \cdots t_n)[u] \). We have \( \eta_\beta(v t_1 \cdots t_n) \leq \eta_\beta(v t_1 \cdots t_n)[u] < \eta_\beta(t) \) so that the first i.h. gives \( v t_1 \cdots t_n = t' \in SN(\beta) \) and thus we are done.

**Lemma 4.6** Let \( t \) be a \( \lambda \)-term. If \( t \in SN(\beta) \), then \( t \in SN(\beta) \).
Proof. By Lem. 4.1 and 4.2 it is sufficient to show that \( t \in ISN(\beta) \) implies \( t \in ISN(\kappa) \).

- If \( t = \ldots t_n \in ISN(\beta) \), where \( t_1, \ldots, t_n \in ISN(\beta) \), then the i.h. gives \( t_1, \ldots, t_n \in ISN(\kappa) \) so that \( t \in ISN(\kappa) \) by rule (H).
- If \( t = \lambda x.v \in ISN(\beta) \), where \( v \in ISN(\beta) \), then the i.h. gives \( v \in ISN(\kappa) \) so that \( \lambda x.v \in ISN(\kappa) \) by rule (A).
- If \( t = (\lambda x.v)u \ldots t_n \in ISN(\beta) \), where \( v(x/u) t_1 \ldots t_n, u \in ISN(\beta) \), then we reason by cases.
  
  Suppose \( x \in \text{fv}(v) \). The i.h. gives \( v(x/u) t_1 \ldots t_n \in ISN(\kappa) \) so that \( t \in ISN(\kappa) \) by rule (I).
  
  Suppose \( x \notin \text{fv}(v) \). The i.h. gives \( vt_1 \ldots t_n, u \in ISN(\kappa) \) so that \( (vt_1 \ldots t_n)[u] \in ISN(\kappa) \) by rule (V), and thus \( t \in ISN(\kappa) \) by rule (G).

The last two lemmas, and the fact the \( \lambda \)-terms are strictly included in \( K \)-terms, allow to conclude:

Corollary 4.7 Let \( t \) be a \( \lambda \)-term. Then \( t \) is \( K \)-typable if and only if \( t \in SN(\beta) \).

5 An Alternative Proof

In this section we give an alternative proof of the characterization result given in Cor. 4.7 which can be found for example in [30,3]. We start by defining a non-erasing \( \beta \)-step as a step of the form \( C[(\lambda x.u)v] \rightarrow_\beta C[u\{x/v\}] \) where \( x \in \text{fv}(u) \).

The following property is a particular case of Thm. 3.4:

Property 1 (Weighted Subject Reduction for Non-Erasing Reductions) Let \( t \) be a \( \lambda \)-term. If \( \Phi \vdash \Gamma \vdash t:\sigma \) and \( t \rightarrow_\beta t' \) is a non-erasing step, then there exists \( \Phi' \vdash \Gamma \vdash t':\tau \) such that \( \text{sz}(\Phi) > \text{sz}(\Phi') \).

Proof. By induction on the reduction relation \( \rightarrow_\beta \) using Lem. 3.3.

Again, notice that the following property is a particular case of Thm. 3.6.

Lemma 5.1 (Subject Expansion for Non-Erasing Reductions) Let \( t' \) be a \( \lambda \)-term. If \( \Phi' \vdash \Gamma \vdash t':\sigma \) and \( t \rightarrow_\beta t' \) is a non-erasing step, then there exists \( \Phi \vdash \Gamma \vdash t:\tau \).

The characterization of \( SN(\beta) \) is then achieved by means of the following arguments:

Theorem 5.2 If \( \Phi \vdash \Gamma \vdash t:\sigma \), then \( t \in SN(\beta) \).

Proof. We use Lem. 4.1 to replace in the statement \( SN(\beta) \) by \( ISN(\beta) \). We proceed by induction on \( \text{sz}(\Phi) \). When \( \Phi \) ends with the rule (ax) or (\( \rightarrow_i \)), the proof is straightforward. Let \( \Phi \) be a derivation ending with (\( \rightarrow_e \)), so that \( t = (\lambda x.u)v \). By construction \( \Gamma = \Delta + j \in J \Gamma_j \) and there are derivations \( \Phi_{x,u} \vdash \Delta \vdash \lambda x.u[\sigma_j]_{j \in I} \rightarrow \sigma \) and \( (\Phi_j \vdash \Gamma_j \vdash v;\sigma_j)_{j \in J} \), where \( I = \emptyset \) implies \( |J| = 1 \) and \( I \neq \emptyset \) implies \( J = I \).

In both cases there is \( j \in J \) such that \( \Gamma_j \vdash v;\sigma_j \) and \( \text{sz}(\Phi_j) < \text{sz}(\Phi) \) so that \( v \in ISN(\beta) \) holds by the i.h. We consider two cases.
• $x \in \text{fv}(u)$. By Lem. 1 we have $\Phi' \vdash \Gamma \vdash u\{x/v\} : \sigma$ and $sz(\Phi') < sz(\Phi)$. Then by the i.h. $u\{x/v\} \in \mathcal{ISN}(\beta)$. This, together with $v \in \mathcal{ISN}(\beta)$ gives $(\lambda x. u)v \in \mathcal{ISN}(\beta)$.

• $x \notin \text{fv}(u)$. Then $I = \emptyset$ and by construction there is $\Phi_u \vdash \Delta \vdash u : \sigma$ verifying $sz(\Phi_u) < sz(\Phi)$, so that $u \in \mathcal{ISN}(\beta)$. This, together with $v \in \mathcal{ISN}(\beta)$ gives $(\lambda x. u)v \in \mathcal{ISN}(\beta)$.

\[ \square \]

**Theorem 5.3** If $t \in \mathcal{SN}(\beta)$, then there exists $\Phi' \vdash \Gamma \vdash t : \sigma$.

**Proof.** We use Lem. 4.1 to replace in the statement $\mathcal{SN}(\beta)$ by $\mathcal{ISN}(\beta)$. We reason by induction $t \in \mathcal{ISN}(\beta)$. The two first cases are straightforward. Let $t = (\lambda x. u)v_1 \ldots v_n \in \mathcal{ISN}(\beta)$ coming from $u\{x/v\}t_1 \ldots t_n, v \in \mathcal{ISN}(\beta)$. By the i.h. $u\{x/v\}t_1 \ldots t_n$ and $v$ are both typable. We consider two cases. If $x \in \text{fv}(u)$, then $(\lambda x. u)v_1 \ldots v_n$ is typable by Lem. 5.1. Otherwise, by construction, we get typing derivations for $u, t_1 \ldots, t_n$ which can easily be used to build a typing derivation of $(\lambda x. u)v_1 \ldots v_n$.

\[ \square \]

### 6 Conclusion

We have presented a characterization of $\beta$-strongly normalizing $\lambda$-terms via a typing system based on non-idempotent intersection types, through an embedding of the $\lambda$-calculus into the memory $K$-calculus.

As a matter of fact, as shown in Section 5, a more direct proof of the same result could be achieved by restricting the system $K$ to the plain lambda calculus: weighted subject reduction is clearly preserved, ensuring that typing implies strong normalization. The other way around is proved by induction on $\mathcal{ISN}(\beta)$, an inductive characterization of the set of $\beta$-strongly normalizing terms.

Still, the good point of the memory calculus is its awareness: nothing is lost along reductions, and therefore subject expansion does hold even for the type system characterizing strong normalization, whereas it does not hold for other calculi, in general. This allows for simple, modular proofs of strong normalization for calculi that can be embedded in a memory-like calculus. In this paper, we show the simplest case, namely the one of the pure $\lambda$-calculus, but extensions to other frameworks, inspired for instance by classical or linear calculi may naturally be conceived.

### References


48


Abstract

We formulate principles of induction and recursion for a variant of lambda calculus in its original syntax (i.e., with only one sort of names) where $\alpha$-conversion is based upon name swapping as in nominal abstract syntax. The principles allow to work modulo $\alpha$-conversion and implement the Barendregt variable convention. We derive them all from the simple structural induction principle on concrete terms and work out applications to some fundamental meta-theoretical results, such as the substitution lemma for $\alpha$-conversion and the lemma on substitution composition. The whole work is implemented in Agda.

Keywords: Formal Metatheory, Lambda Calculus, Constructive Type Theory

1 Introduction

We are interested in methods for formalising in constructive type theory the meta-theory of the lambda-calculus. The main reason for this is that the lambda calculus is both a primigenial programming language and a prime test bed for formal reasoning on tree structures that feature (name) binding.

Selected, revised papers will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
Specifically concerning the latter, the informal procedure consists to begin with in "identifying terms up to \( \alpha \)-conversion". However, this is not simply carried out when functions are defined by recursion and properties proven by induction. The problem has to do with the fact that the consideration of the \( \alpha \)-equivalence classes is actually conducted through the use of convenient representatives thereof. These are chosen by the so-called Barendregt Variable Convention (BVC): each term representing its \( \alpha \)-class is assumed to have bound names all different and different from all names free in the current context. Now, a general validity criterion determines that this procedure ought to be accompanied in all cases by the verification that the proofs and results of functions depend only on the \( \alpha \)-class and do not vary with the particular choice of the representative in question. Such verification is seldom accomplished but yet it is not the main difficulty concerning the validity of the constructions so performed. The crucial point is that e.g. inductive proofs are often carried out employing the structural principle for concrete terms — and then it may well happen that an induction step corresponding to functional abstractions can be carried out for a conveniently chosen bound name but not for an arbitrary one as the principle requires.

The problem can be avoided by the use of de Bruijn’s nameless syntax [4] or its more up-to-date version locally nameless syntax [2,3], which uses names for the free or global variables and the indices counting up to the binding abstractor for the occurrences of local parameters. But these methods are not without overhead in the form of several operations or well-formedness predicates. As a result, there certainly is a relief in not having to consider \( \alpha \)-conversion; but, at the same time, the nameless syntax seriously affects the connection between actual formal procedures and what could be considered the natural features of syntax. The same has to be said of the map representation introduced in [9].

A different alternative is to replace the (as explained above, problematic) use of structural induction and recursion principles on concrete terms by that of so-called \( \alpha \)-structural principles working directly on the \( \alpha \)-equivalence classes. This means providing principles that allow to prove properties by induction and to define functions by recursion by direct use of the BVC, so as to ease the burden associated to the verification of the validity of the procedure.

A first attempt in this direction is [6], which gives an axiomatic description of lambda terms in which equality embodies \( \alpha \)-conversion and that provides a method of definition of functions by recursion on such type of objects. This work ultimately rests upon the use of higher-order abstract syntax within the HOL system, and a theoretical model using de Bruijn’s nameless syntax is sketched to show the soundness of the system of axioms. In [5,12,13], models of syntax with binders are introduced which formulate the basic concepts of abstraction, \( \alpha \)-equivalence and a name being “sufficiently fresh” in a mathematical object, on the basis of the simple operation of name swapping. This theory — which has become known as nominal abstract syntax — provides a framework of (first-order) languages with binding with associated principles of \( \alpha \)-structural recursion and induction that are based on the verification of the non-dependence of the mathematical objects in the current context, as well as of the results of step functions used in recursive definitions, on the bound names chosen for the representatives of the \( \alpha \)-classes involved. Implementations of this
Copello, Tasistro, Szasz, Bove, and Fernandez

approach have been tried in Isabelle/HOL [15] and Coq [1]. In the first case the solution rests upon a weak version of higher-order abstract syntax, whereas the second one is an axiomatisation in which —similarly to [6] cited above— equality is postulated as embodying $\alpha$-conversion and a model of the system based on locally nameless syntax has been constructed.

Yet another approach to the formulation of the alpha-structural principles originates in the observation that, if the property to be tried is $\alpha$-compatible —i.e., it is actually a property of the $\alpha$-classes and not just of the concrete terms— then (complete) induction on the size of terms can be used to bridge over the possible gap pointed out above in proofs by induction that confine themselves to convenient choice of bound names. Indeed, suppose you need to prove $P(\lambda x.M)$; now, if what you have is a step from $P(M^*)$ to $P(\lambda x^*.M^*)$ for a convenient renaming of the term, then you will be able to use your strong size-induction hypothesis on $M^*$, since this is still of a size lesser than that of $P(\lambda x.M)$. Hence you will arrive at $P(\lambda x^*.M^*)$ and from there to the desired $P(\lambda x.M)$ because of the $\alpha$-compatibility of $P$. This motivates trying to provide a mechanism of this kind to formalise the use of the BVC, and that is what we attempt in this paper. The result is that we are able to provide principles of alpha-structural induction and recursion, implementing the BVC in constructive type theory, using just the ordinary first-order, name-carrying syntax and actually without using the strong induction on the size of the terms —i.e. we are able to derive the principles in question from just simple structural induction on concrete terms. To such effect we define $\alpha$-equivalence by using the basic concepts of nominal abstract syntax, namely freshness and swapping of names. Equality remains the simple definitional one and we do not either perform any kind of quotient construction. The whole development is implemented in the Agda system [10].

The rest of the paper goes as follows: in section 2 we present the infrastructure just mentioned. Section 3 presents the principles, starting from the simple structural induction on terms and ending up with the recursion principle on $\alpha$-classes. In section 4 we show several applications that bring about certain feeling for the usefulness of the method. Finally, section 5 compares with related work and points out conclusions and further work.

The present is actually a literate Agda document, where we hide some code for reasons of conciseness. The entire code is available at:

https://github.com/ernius/formalmetatheory-nominal

and has been compiled with the last Agda version 2.4.2.2 and 0.9 standard library.

2 Infrastructure

2.1 Agda

Agda implements Constructive Type Theory [8] (type theory for short). It is actually a functional programming language in which:

(i) Inductive types can be introduced as usual, i.e. by enumeration of their constructors, but they can be parameterised in objects of other types. Because of the latter it is said that type theory features families of types (indexed by a
base type) or dependent types.

(ii) Functions on families of types respect the dependence on the base object, which is to say that they are generally of the form \((x : \alpha) \to \beta_x\) where \(\beta_x\) is the type parameterised on \(x\) of type \(\alpha\). Therefore the type of the output of a function depends on the value of the input.

(iii) Functions on inductive types are defined by pattern-matching equations.

(iv) Every function of the language must be terminating. The standard form of recursion that forces such condition is structural recursion and is, of course, syntactically checked.

(v) Because of the preceding feature, type theory can be interpreted as a constructive logic. Specifically, this is achieved by representing propositions as inductive types whose constructors are the introduction rules, i.e. methods of direct proof, of the propositions in question.

Therefore we can say in summary that sets of data, predicates and relations are defined inductively, i.e. by enumeration of their constructors.

2.2 Syntax

The set \(\Lambda\) of terms is as usual. It is built up from a denumerable set of names, which we shall call atoms, borrowing terminology from nominal abstract syntax.

\[
\text{data } \Lambda : \text{Set where}
\]

\[
\begin{align*}
\nu : & \text{Atom} \to \Lambda \\
\_\cdot_ : & \Lambda \to \Lambda \to \Lambda \\
\chi : & \text{Atom} \to \Lambda \to \Lambda \\
\end{align*}
\]

The following is called the freshness relation. It holds when a variable does not occur free in a term. Parameters to a function written between curly brackets can be omitted when invoking the function.

\[
\text{data } _\#_ : (a : \text{Atom}) : \Lambda \to \text{Set where}
\]

\[
\begin{align*}
\#\nu : & \{b : \text{Atom}\} \to b \not= a \to a \# \nu b \\
\#\cdot : & \{M N : \Lambda\} \to a \# M \to a \# N \to a \# M : N \\
\#\chi : & \{M : \Lambda\} \to a \# \chi a M \\
\#\chi : & \{b : \text{Atom}\} \{M : \Lambda\} \to a \# M \to a \# \chi b M \\
\end{align*}
\]

Next comes the fundamental operation of swapping of atoms. A finite sequence (composition) of atom swaps constitutes a (finite) atom permutation which is the renaming mechanism to be used on terms. The action of atom swaps is first defined on atoms themselves:

\[
\begin{align*}
(\_\cdot\_)_a : & \text{Atom} \to \text{Atom} \to \text{Atom} \\
(a \cdot b)_a c \ & \text{with} \ c \not\approx_a a \\
& \text{if} \ \yes \ _ = b \\
& \text{if} \ \no \ _ = \chi \ a b \\
& \text{if} \ \not\approx_a c = a \\
& \text{if} \ \not\approx_a c = c \\
\end{align*}
\]
Here it extends to terms:

\[ (_\cdot_\cdot_) : \text{Atom} \rightarrow \text{Atom} \rightarrow \Lambda \rightarrow \Lambda \]
\[ (a \cdot b) \cdot c = (\lambda (a \cdot b) \cdot c) \]
\[ (a \cdot b) \cdot M \cdot N = ((a \cdot b) \cdot M) \cdot ((a \cdot b) \cdot N) \]
\[ (a \cdot b) \cdot \chi c \cdot M = \chi ((a \cdot b) \cdot c) ((a \cdot b) \cdot M) \]

And the same goes for permutations, which are lists of swaps:

\[ _\cdot_\cdot a : \Pi \rightarrow \text{Atom} \rightarrow \text{Atom} \]
\[ \pi \cdot a = \text{foldr} (\lambda s b \cdot (\text{proj}_1 s \cdot \text{proj}_2 s) \cdot a) \cdot a \cdot \pi \]

\[ _\cdot_\cdot M = \text{foldr} (\lambda s M \cdot (\text{proj}_1 s \cdot \text{proj}_2 s) \cdot M) \cdot M \cdot \pi \]

We now introduce \(\alpha\)-conversion, denoted by \(\sim\alpha\). We use a syntax-directed definition that uses co-finite quantification in the case of the lambda abstractions:

\[
\text{data } _\sim\alpha_ : \Lambda \rightarrow \Lambda \rightarrow \text{Set} \text{ where }
\sim\alpha v : \{a : \text{Atom}\} \rightarrow v \sim\alpha v a
\sim\alpha : \{M M' N N' : \Lambda\} \rightarrow M \sim\alpha M' \rightarrow N \sim\alpha N'
\rightarrow M \cdot N \sim\alpha M' \cdot N'
\sim\alpha\chi : \{M N : \Lambda\} \cdot \{a b : \text{Atom}\} (xs : \text{List Atom})
\rightarrow ((c : \text{Atom}) \rightarrow c \notin xs \rightarrow (a \cdot c) \sim\alpha (b \cdot c) N)
\rightarrow \chi a M \sim\alpha \chi b N
\]

The idea is that for proving two abstractions \(\alpha\)-equivalent you should be able to prove the respective bodies \(\alpha\)-equivalent when you rename the bound names to any name not free in both abstractions. The condition on the new name can be generalised to “any name not in a given list”, yielding an equivalent relation. The latter condition is harder to prove, but more convenient to use when you assume \(\sim\alpha\) to hold, which is more often the case in the forthcoming proofs.

### 3 Alpha-Structural Induction and Recursion Principles

We start with the simple structural induction over the concrete \(\Lambda\) terms:

\[
\text{TermPrimInd} : \{l : \text{Level}\} (P : \Lambda \rightarrow \text{Set} l)
\rightarrow (\forall a \rightarrow P (v a))
\rightarrow (\forall M N \rightarrow P M \rightarrow P N \rightarrow P (M \cdot N))
\rightarrow (\forall M b \rightarrow P M \rightarrow P (\chi b M))
\rightarrow \forall M \rightarrow P M
\]

Fig. 1. Concrete Structural Induction Principle

The next induction principle provides a strong hypothesis for the lambda abstraction case: it namely allows to assume the property for all renamings (given by finite permutations of names) of the body of the abstraction:

Notice that the hypothesis provided for the case of abstractions is akin to the corresponding one of the principle of strong or complete induction on the size of terms,
TermIndPerm : \{l : Level\}\{P : \Lambda \rightarrow Set l\}
\rightarrow (\forall a \rightarrow P (\forall a))
\rightarrow (\forall M N \rightarrow P M \rightarrow P N \rightarrow P (M \cdot N))
\rightarrow (\forall M b \rightarrow (\forall \pi \rightarrow P (\pi \bullet M)) \rightarrow P (\lambda b M))
\rightarrow \forall M \rightarrow P M

Fig. 2. Strong Permutation Induction Principle

only that expressed in terms of name permutations. This principle can be derived from the former, i.e. from simple structural induction, in very much the same way as complete induction on natural numbers is derived from ordinary mathematical induction. That is to say, we can use structural induction to prove \((\forall \pi)P(\pi \bullet M)\) given the hypotheses of the new principle, from which \(PM\) follows. For the interesting case of abstractions, we have to prove \((\forall \pi)P(\pi \bullet a M)\), which is equal to \((\forall \pi)P(\lambda (\pi \bullet a) (\pi \bullet M))\). The hypothesis of the new principle give us in this case \((\forall M', b)((\forall \pi')P(\pi' \bullet M') \rightarrow P(\lambda b M'))\). Now, instantiating \(M'\) as \(\pi \bullet M\) and \(b\) as \(\pi \bullet a\), we obtain the desired result if we know that \((\forall \pi')P(\pi' \bullet \pi \bullet M)\), which holds by induction hypothesis of the structural principle.

We call a predicate \(\alpha\)-compatible if it is preserved by \(\alpha\)-conversion:
\[\alphaCompatPred : \{l : Level\}\rightarrow (\Lambda \rightarrow Set l) \rightarrow Set l\]
\[\alphaCompatPred P = \{M N : \Lambda\} \rightarrow M \sim_\alpha N \rightarrow P M \rightarrow P N\]

For \(\alpha\)-compatible predicates we can use the preceding principle to derive the following:
\[Term\alphaPrimInd : \{l : Level\}\{P : \Lambda \rightarrow Set l\} \rightarrow \alphaCompatPred P\]
\[\rightarrow (\forall a \rightarrow P (\forall a))\]
\[\rightarrow (\forall M N \rightarrow P M \rightarrow P N \rightarrow P (M \cdot N))\]
\[\rightarrow \exists (\lambda vs \rightarrow (\forall M b \rightarrow b \notin vs \rightarrow P M \rightarrow P (\lambda b M)))\]
\[\rightarrow \forall M \rightarrow P M\]

This new principle enables us to carry out the proof of the abstraction case by choosing a bound name different from the names in a given list \(vs\). It gives a way to emulate the Barendregt Variable Convention (BVC) since, indeed, the names to be avoided will always be finitely many; in using the principle we must provide a list that includes them. This same principle is provided in [1], only that we here give it a proof in terms of the ones previously introduced, instead of just postulating it. Our aim is to employ this principle whenever possible, thereby hiding the use of the swap operation which is confined to the previous principles exposed. The interesting case in the implementation of the principle is of course that of the functional abstraction.

We must put ourselves in the position in which we are using the former strong principle and are given an abstraction \(\lambda b M\) for which we have to prove \(P\). We have to employ to this effect the clause of our new principle corresponding to the functional abstractions, which forces us to employ a name \(b^*\) out of the given list \(vs\). Therefore we can aspire at proving \(P\) for a renaming of the original term, say \(\lambda b^* M^*\). The required result will then follow from the \(\alpha\)-compatibility of the predicate \(P\) provided \(\lambda b^* M^* \sim_\alpha \lambda b M\). This imposes the condition that the name
$b^*$ be chosen fresh in the original term $\lambda b M$ —and that $M^* = (b^* \bullet b) M$. We know $PM^*$ and therefore $P(\lambda b^* M^*)$ because we know $P$ for any renaming of $M$, by the hypothesis of the strong principle from which we start.

A very important point in this implementation is that, given the list of names to be avoided, we can and do choose $b^*$ deterministically for each class of $\alpha$-equivalent terms. Indeed, if we determine $b^*$ as e.g. the first name out of the given list that is fresh (i.e. not free) in the originally given term, then the result will be one and the same for every term of each $\alpha$-class, since $\alpha$-equivalent terms have the same free variables. Hence the representative of each $\alpha$-class chosen by this method will be fixed for each list of names to be avoided, which constitutes a basis for using the method for defining functions on the $\alpha$-classes. This will work by associating to (each term of) the class the result of the corresponding computation on the canonically chosen representative.

More precisely, let us say that a function $f : \Lambda \to A$ is strongly $\alpha$-compatible iff $M \sim_\alpha N \Rightarrow fM = fN$. We can now define an iteration principle over raw terms which always produces strongly $\alpha$-compatible functions. For the abstraction case, this principle also allows us to give a list of variables from where the abstractions variables are not to be chosen. This iteration principle is derived from the BVC induction principle ($\text{Term} \alpha \text{PrimInd}$) in a direct manner, just using a trivial constant predicate equivalent to the type $A$. We exhibit the type and code of the iterator:

$$\Lambda \text{Alt} : \{l : \text{Level}\}(A : \text{Set } l)$$
$$\to (\text{Atom } \to A)$$
$$\to (A \to A \to A)$$
$$\to \text{List Atom } \times (\text{Atom } \to A \to A)$$
$$\to \Lambda \to A$$

$$\Lambda \text{Alt } A \text{ hv } \cdot (\text{vs } , \text{ h } k)$$

$$= \text{Term} \alpha \text{PrimInd} \ (\lambda _ \_ \to A)$$

$$\quad (\lambda _ \_ \to \text{id})$$

$$\quad \text{hv}$$

$$\quad (\lambda _ \_ \_ \to h \cdot)$$

$$\quad (\text{vs } , (\lambda _ b _ r \to h \cdot b r))$$

To repeat the idea, the iterator works as a function on $\alpha$-classes because for each given abstraction, it will yield the result obtained by working on a canonically chosen representative that is determined by the list of names to be avoided and the (free names of the) $\alpha$-class in question. Strong compatibility would not obtain if we tried directly to formulate a recursion instead of an iteration principle, but we can recover the more general form by the standard procedure of computing pairs one of whose components is a term. Thereby we arrive at the next recursion principle over terms, which also generates strong $\alpha$-compatible functions.

$$\Lambda \text{Rec} : \{l : \text{Level}\}(A : \text{Set } l)$$
$$\to (\text{Atom } \to A)$$
$$\to (A \to A \to \Lambda \to \Lambda \to A)$$
$$\to \text{List Atom } \times (\text{Atom } \to A \to \Lambda \to A)$$
$$\to \Lambda \to A$$
4 Applications in Meta-Theory

We present several applications of the iteration/recursion principle defined in the preceding section. In the following two sub-sections we implement two classic examples of λ-calculus theory. In the appendix A we also apply our iteration/recursion principle to the examples of functions over terms presented in [11]. This work presents a sequence of increasing complexity functions, with the purpose of testing the applicability of recursion principles over λ-calculus terms.

4.1 Free Variables

We implement the function that returns the free variables of a term.

\[ \text{fv} : \Lambda \rightarrow \text{List Atom} \]

\[ \text{fv} = \text{Alt (List Atom)} \_++\_ (\[\_\], \lambda v r \rightarrow r \cdot v) \]

As a direct consequence of strong \(\alpha\)-compatibility of the iteration principle we have that \(\alpha\)-equivalent terms have the same free variables.

The relation \(\_\_\_\_\) holds when a variable occurs free in a term.

\[ \text{data } \_\_\_\_ : \text{Atom} \rightarrow \Lambda \rightarrow \text{Set} \]

\[ \_\_\_\_ : \{x : \text{Atom}\} \rightarrow x \cdot \] 

\[ \_\_\_\_ : \{x : \text{Atom}\}\{M N : \Lambda\} \rightarrow x \cdot M \rightarrow x \cdot (M \cdot N) \]

\[ \_\_\_\_ : \{x y : \text{Atom}\}\{M : \Lambda\} \rightarrow x \cdot M \rightarrow y \neq x \rightarrow x \cdot (\lambda y M) \]

We can use our BVC-like induction principle to prove the following proposition:

\[ \text{Pfv}^* : \text{Atom} \rightarrow \Lambda \rightarrow \text{Set} \]

\[ \text{Pfv}^* a M = a \in \text{fv} M \rightarrow a \cdot M \]

In the case of lambda abstractions we are able to simplify the proof by choosing the bound name different from \(a\). This flexibility comes at a cost, i.e. we need to prove that the predicate \(\text{Pfv}^* a\) is \(\alpha\)-compatible in order to use the chosen induction principle. This \(\alpha\)-compatibility proof is direct once we prove that \(\_\_\_\_\) is an \(\alpha\)-compatible relation and the \(\text{fv}\) function is strong \(\alpha\)-compatible. The last property is direct because we implemented \(\text{fv}\) with the iteration principle, so the extra cost is just the proof that \(\_\_\_\_\) is \(\alpha\)-compatible. This in turn could be directly obtained if we defined the relation establishing that a variable \(a\) is free in a term as a recursive function, as follows:

\[ \_\_\_ \_ : \text{Atom} \rightarrow \Lambda \rightarrow \text{Set} \]

\[ (\_\_\_ \_ ) \_ = \Lambda \_ \text{Set} (\lambda b \rightarrow a \equiv b) \_\_\_\_ (\[\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] ) \]

For the variable case we return the propositional equality of the searched variable to the term variable. The application case is the disjoint union of the types returned by the recursive calls. Finally, in the abstraction case we can choose the abstraction variable to be different from the searched one. In this way we can ignore the abstraction variable and return just the recursive call containing the evidence of any free occurrence of the searched variable in the abstraction body. This implementation is strong compatible by construction because we have built it from our iterator principle, so it is also immediate from this definition that \(\alpha\)-equivalent terms have the same free variables.
4.2 Substitution

We implement capture avoiding substitution in the following way:

\[
\text{hvar} : \text{Atom} \rightarrow \Lambda \Rightarrow \text{Atom} \rightarrow \Lambda
\]

\[
hvar \ x \ y \ \text{with} \quad x \overset{?}{=} a \quad y
\]

\[
\ldots \mid \text{yes} \ _{\_} = N
\]

\[
\ldots \mid \text{no} \ _{\_} = v \ y
\]

\[
\Delta[\_] : \Lambda \rightarrow \text{Atom} \rightarrow \Lambda \rightarrow \Lambda
\]

\[
M \ [\ a := \ N] = \text{Alt} \ (hvar \ a \ N) \ _{\_} \ (a :: \text{fv} \ N, \ \lambda) \ M
\]

It shows to be quite close to the simple pencil-and-paper version assuming the BVC. Notice that we explicitly indicate that the bound name of the canonical representative to be chosen must be different from the replaced variable and not occur free in the substituted term. Again because of the strong \(\alpha\)-compatibility of the iteration principle we obtain the following result for free:

\[
\text{lemmaSubst1} : \ {M \ N : \ \Lambda} \ (P : \ \Lambda)(a : \ \text{Atom})
\]

\[
\rightarrow M \sim_{\alpha} N \rightarrow M \ [\ a := \ P] \equiv N \ [\ a := \ P]
\]

Using the induction principle in figure 2 we prove:

\[
\text{lemmaSubst2} : \forall \ {\{N\} \ {P\} \ M \ x}
\]

\[
\rightarrow N \sim_{\alpha} P \rightarrow M \ [\ x := \ N] \sim_{\alpha} M \ [\ x := \ P]
\]

From the two previous results we directly obtain the \(\alpha\)-substitution lemma:

\[
\text{lemmaSubst} : \ {M \ N \ P \ Q : \ \Lambda} (a : \ \text{Atom})
\]

\[
\rightarrow M \sim_{\alpha} N \rightarrow P \sim_{\alpha} Q
\]

\[
\rightarrow M \ [\ a := \ P] \sim_{\alpha} N \ [\ a := \ Q]
\]

\[
\text{lemmaSubst} \ {M} \ {N} \ {P} \ {Q} \ a \ M \sim_{\alpha} N \sim_{\alpha} Q
\]

\[
= \begin{align*}
\text{begin} \ M \ [\ a := \ P] \\
\approx \langle \text{lemmaSubst1} \ P \ a \ M \sim_{\alpha} N \rangle \\
N \ [\ a := \ P] \\
\sim \langle \text{lemmaSubst2} \ N \ a \ P \sim_{\alpha} Q \rangle \\
N \ [\ a := \ Q]
\end{align*}
\]

\[
\square
\]

In turn, with the preceding result we can derive that our substitution operation is \(\alpha\)-equivalent with a naïve one for fresh enough bound names:

\[
\text{lemma}^{\alpha}_{\sim} : \forall \ {\{a \ b \ P\} \ M \rightarrow b \notin a :: \text{fv} \ P}
\]

\[
\rightarrow \lambda \ b \ M \ [\ a := \ P] \sim_{\alpha} \lambda \ b \ (M \ [\ a := \ P])
\]

We can combine this last result with the \text{Term}^{\alpha}_{\text{PrimInd}} principle which emulates BVC convention, and mimic in this way pencil-and-paper inductive proofs over \(\alpha\)-equivalence classes of terms about substitution operation. As an example we show
next the substitution composition lemma:

\[ \text{PSC} : \forall \{x\ y\ L\} \ N \rightarrow \Lambda \rightarrow \text{Set} \]
\[ \text{PSC} \{x\} \{y\} \{L\} \ N = x \not\equiv y \rightarrow x \notin \text{fv} \ L \]
\[ \rightarrow (M \{x := N\} \{y := L\} \sim_{\alpha} (M \{y := L\})\{x := N[y := L]\}) \]

We first give a direct equational proof that PSC predicate is \(\alpha\)-compatible:

\[ \alpha\text{CompatiblePSC} : \forall \{x\ y\ L\} \ N \rightarrow \alpha\text{CompatiblePred} \ (\text{PSC} \{x\} \{y\} \{L\} \ N) \]
\[ \alpha\text{CompatiblePSC} \{x\} \{y\} \{L\} \ N \{M\} \{P\} \ M \sim_{\alpha} P \ PM \ x \neq y \ x \notin \text{fv} \ L \]
\[ = \begin{align*}
\text{begin} & \\
(P \{x := N\} \{y := L\}) & \\
\text{Strong } \alpha \text{ compatibility of inner substitution operation} & \\
\cong (\text{cong} ((\lambda\ z \rightarrow z) \{y := L\}) \ (\text{lemmaSubst1} \ N x (\sigma M \sim P))) & \\
(M \{x := N\} \{y := L\}) & \\
\text{We apply that we know the predicate holds for } M & \\
\cong (\text{cong} ((\lambda\ z \rightarrow z) \{x := N \{y := L\}\}) \ (\text{lemmaSubst1} \ L y (M \sim P)) & \\
(P \{y := L\}) \{x := N \{y := L\}\}) & \\
\end{align*} \]

\[ \square \]

For the interesting abstraction case of the \(\alpha\)-structural induction over the lambda term, we assume the abstraction variables in the term are not among the replaced variables or free in the substituted terms. In this way the substitution operations become \(\alpha\)-compatible to naïve substitutions, and the induction hypothesis allows us to complete the the inductive proof in a direct manner. The code fragment becomes:

\[ \text{begin} \]
\[ (\chi\ b\ M \{x := N\} \{y := L\}) \]
\[ \text{Inner substitution is } \alpha \text{ equivalent} \]
\[ \text{to a naive one because } b \notin x \:: \text{fv} \ N \]
\[ \cong (\text{lemmaSubst1} \ L y (\text{lemma}\sim\[] \ M b \not\in x :: \text{fv} N)) \]
\[ (\chi\ b\ (M \{x := N\}) \{y := L\}) \]
\[ \text{Outer substitution is } \alpha \text{ equivalent} \]
\[ \text{to a naive one because } b \notin y :: \text{fv} \ L \]
\[ \cong (\text{lemma}\sim\[\chi\] (\text{IndHip} x \neq y \ x \notin \text{fv} L)) \]
\[ \chi\ b\ ((M \{y := L\}) \{x := N \{y := L\}\}) \]
\[ \text{Outer substitution is } \alpha \text{ equivalent} \]
\[ \text{to a naive one because } b \notin x :: \text{fv} N \{y := L\} \]
\[ \cong (\sigma (\text{lemma}\sim\[\chi\]) \ (M \{y := L\} b \not\in x :: \text{fv} N [y := L])) \]
\[ (\chi\ b\ (M \{y := L\}) \{x := N \{y := L\}\}) \]
\[ \text{Inner substitution is } \alpha \text{ equivalent} \]
\[ \text{to a naive one because } b \notin y :: \text{fv} \ L \]
\[ \cong (\text{sym} (\text{lemmaSubst1} \ (N \{y := L\}) \ x \ (\text{lemma}\sim\[\chi\] \ M b \not\in y :: \text{fv} L))) \]
Remarkably these results are directly derived from the first primitive induction principle, and no induction on the length of terms or accessible predicates were needed in all of this formalization.

5 Conclusions

The main contribution of this work is a full implementation in Constructive Type Theory of principles of induction and recursion allowing to work on \( \alpha \)-classes of terms of the lambda calculus. The crucial component seems to be what we called a BVC-like induction principle allowing to choose the bound name in the case of the abstractions so that it does not belong to a given list of names. This principle is, on the one hand, derived (for \( \alpha \)-compatible predicates) from ordinary structural induction on concrete terms, thus avoiding any form of induction on the size of terms or other more complex forms of induction. And, on the other hand, it gives rise to principles of recursion that allow to define functions on \( \alpha \)-classes, specifically, functions giving identical results for \( \alpha \)-equivalent terms. We have also shown by way of a number of examples that the principles provide a flexible framework quite able to pleasantly mimic pencil-and-paper practice.

Our work departs from e.g. [13] in that we do fix the choice of representatives for implementing the alpha-structural recursion thereby forcing this principle to yield identical results for \( \alpha \)-equivalent terms. This might be a little too concrete but, on the other hand, it gives us the possibility of completing a simple full implementation on an existing system, as different from other works which base themselves on postulates or more sophisticated systems of syntax or methods of implementation.

We wish to continue exploring the capabilities of this method of formalisation by studying its application to the meta-theory of type systems. We also wish to deepen its comparison to the method based on Stoughton’s substitutions [14], which we started to investigate in [7] and which we believe can give rise to formulations similar to the one exposed here.

A Iteration/Recursion Applications

In the following sections we successfully apply our iteration/recursion principle to all the examples from [11]. This work presents a sequence of functions whose definitions are increasing in complexity to provide a test for any principle of function definition, where each of the given functions respects the \( \alpha \)-equivalence relation.

A.1 Case Analysis and Examining Constructor Arguments

The following family of functions distinguishes between constructors returning the constructor components, giving in a sense a kind of pattern-matching.
\[
\begin{align*}
\text{isVar} : \Lambda \rightarrow & \quad \text{Maybe (Variable)} & \\
isVar (v x) = & \quad \text{Just} & \\
isVar (M \cdot N) = & \quad \text{Nothing} & \\
isVar (\lambda x M) = & \quad \text{Nothing} & \\
\text{isApp} : \Lambda \rightarrow & \quad \text{Maybe (\Lambda \times \Lambda)} & \\
isApp (v x) = & \quad \text{Nothing} & \\
isApp (M \cdot N) = & \quad \text{Just}(M, N) & \\
isApp (\lambda x M) = & \quad \text{Nothing} & \\
\text{isAbs} : \Lambda \rightarrow & \quad \text{Maybe (Variable \times \Lambda)} & \\
isAbs (v x) = & \quad \text{Nothing} & \\
isAbs (M \cdot N) = & \quad \text{Nothing} & \\
isAbs (\lambda x M) = & \quad \text{Just}(x, M) & \\
\end{align*}
\]

Next we present the corresponding encodings into our iteration/recursion principle:

\[
\begin{align*}
isVar : \Lambda \rightarrow & \quad \text{Maybe Atom} & \\
isVar = & \quad \Lambda \text{It (Maybe Atom)} & \\
& \quad \text{just} & \\
& \quad (\lambda _ _ \rightarrow \text{nothing}) & \\
& \quad ([], \lambda _ _ \rightarrow \text{nothing}) & \\
\end{align*}
\]

\[
\begin{align*}
isApp : \Lambda \rightarrow & \quad \text{Maybe (\Lambda \times \Lambda)} & \\
isApp = & \quad \Lambda \text{Rec (Maybe (\Lambda \times \Lambda))} & \\
& \quad (\lambda _ \rightarrow \text{nothing}) & \\
& \quad (\lambda _ _ M N \rightarrow \text{just} (M, N)) & \\
& \quad ([], \lambda _ _ _ \rightarrow \text{nothing}) & \\
\end{align*}
\]

\[
\begin{align*}
isAbs : \Lambda \rightarrow & \quad \text{Maybe (Atom \times \Lambda)} & \\
isAbs = & \quad \Lambda \text{Rec (Maybe (Atom \times \Lambda))} & \\
& \quad (\lambda _ \rightarrow \text{nothing}) (\lambda _ _ _ \rightarrow \text{nothing}) & \\
& \quad ([], \lambda a _ M \rightarrow \text{just} (a, M)) & \\
\end{align*}
\]

\section{Simple recursion}

The size function returns a numeric measurement of the size of a term.

\[
\begin{align*}
\text{size} : \Lambda \rightarrow & \quad \mathbb{N} & \\
\text{size} (v x) = & \quad 1 & \\
\text{size} (M \cdot N) = & \quad \text{size}(M) + \text{size}(N) + 1 & \\
\text{size} (\lambda x M) = & \quad \text{size}(M) + 1 & \\
\end{align*}
\]

\[
\begin{align*}
\text{size} : \Lambda \rightarrow & \quad \mathbb{N} & \\
\text{size} = & \quad \Lambda \text{lt } \mathbb{N} (\text{const 1}) (\lambda n m \rightarrow \text{suc } n + m) (\, [], \lambda _ n \rightarrow \text{suc } n) & \\
\end{align*}
\]
A.3 Alpha Equality

This function decides the α-equality relation between two terms.

\[
\text{equal} : \Lambda \to \Lambda \to \text{Bool}
\]

\[
\text{equal} = \forall t (\Lambda \to \text{Bool}) \text{ vareq appeq } ([], \text{abseq})
\]

where

\[
\text{vareq} : \text{Atom} \to \Lambda \to \text{Bool}
\]

\[
\text{vareq} a M \text{ with isVar } M
\]

... | nothing = false

... | just \( b \) = \[ a \overset{?}{=} b \]

\[
\text{appeq} : (\Lambda \to \text{Bool}) \to (\Lambda \to \text{Bool}) \to \Lambda \to \text{Bool}
\]

\[
\text{appeq } fM fN P \text{ with isApp } P
\]

... | nothing = false

... | just \( (M', N') = fM M' \land fN N' \)

\[
\text{abseq} : \text{Atom} \to (\Lambda \to \text{Bool}) \to \Lambda \to \text{Bool}
\]

\[
\text{abseq } a fM N \text{ with isAbs } N
\]

... | nothing = false

... | just \( (b, P) = [ a \overset{?}{=} b ] \land fM P \)

Observe that \( \text{isAbs} \) function also normalises \( N \), so it is correct in the last line to ask if the two bound names are the same.

A.4 Recursion Mentioning a Bound Variable

The \( \text{enf} \) function is true of a term if it is in \( \eta \)-normal form. It invokes the \( \text{fv} \) function, which returns the set of a term’s free variables and was previously defined.

\[
\text{enf} : \Lambda \rightarrow \text{Bool}
\]

\[
\text{enf} (v x) = \text{True}
\]

\[
\text{enf} (M \cdot N) = \text{enf}(M) \land \text{enf}(N) + 1
\]

\[
\text{enf} (\lambda x M) = \text{enf}(M) \land (\exists N, x/\text{isApp}(M) \Rightarrow \text{Just}(N, v x) \Rightarrow x \in \text{fv}(N))
\]

\[
\Rightarrow : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}
\]

\[
\text{false} \Rightarrow b = \text{true}
\]

\[
\text{true} \Rightarrow b = b
\]

\[
\text{enf} : \Lambda \rightarrow \text{Bool}
\]

\[
\text{enf} = \Lambda \text{Rec } \text{Bool} \text{ (const true) } (\lambda b1 b2 \_ \_ \rightarrow b1 \land b2) ([], \text{absenf})
\]

where

\[
\text{absenf} : \text{Atom} \rightarrow \text{Bool} \rightarrow \Lambda \rightarrow \text{Bool}
\]

\[
\text{absenf } a b M \text{ with isApp } M
\]

... | nothing = b

... | just \( (P, Q) = b \land (\text{equal } Q (v a) \Rightarrow a \in b \text{ (fv } P)) \)
A.5 Recursion with an Additional Parameter

Given the ternary type of possible directions to follow when passing through a term \((Lt, Rt, In)\), corresponding to the two sub-terms of an application constructor and the body of an abstraction, return the set of paths (lists of directions) to the occurrences of the given free variable in a term. Assume \textit{cons} insert an element in front of a list.

\[
vposns : \text{Variable} \times \Lambda \rightarrow \text{List} (\text{List Direction})
\]

\[
vposns (x, v y) = \begin{cases} & \text{if } (x == y) \text{ then } [[]] \text{ else } [] \\ & \text{else} \\ & \text{map (cons Lt) (vposns x M)} \maplus \text{map (cons Rt) (vposns x N)} \end{cases}
\]

\[x \neq y \Rightarrow vposns (x, \lambda y M) = \text{map (cons In) (vposns x M)}\]

Notice how the condition guard of the abstraction case is translated to the list of variables from where not to choose the abstraction variable.

\[
data \text{ Direction } : \text{Set} \ where \\
\text{Lt, Rt, In} : \text{Direction}
\]

\[
vposns : \text{Atom} \rightarrow \Lambda \rightarrow \text{List} (\text{List Direction})
\]

\[
vposns a = \Lambda \text{lt} (\text{List (List Direction)}) \text{ varvposns appvposns ([ a ], absvposns)}
\]

where

\[
\text{varvposns : Atom} \rightarrow \text{List (List Direction)}
\]

\[
\text{varvposns b with a \equiv_a b}
\]

\[
... | \text{yes } a = [] \\
... | \text{no } a = []
\]

\[
\text{appvposns : List (List Direction)} \rightarrow \text{List (List Direction)}
\]

\[
\rightarrow \text{List (List Direction)}
\]

\[
\text{appvposns l r} = \text{map (cons Lt) l} \maplus \text{map (cons Rt) r}
\]

\[
\text{absvposns : Atom \rightarrow List (List Direction) \rightarrow List (List Direction)}
\]

\[
\text{absvposns a r} = \text{map (cons In) r}
\]

A.6 Recursion with Varying Parameters and Terms as Range

A variant of the substitution function, which substitutes a term for a variable, but further adjusts the term being substituted by wrapping it in one application of the variable named "0" per traversed binder.
sub′ : Λ × Variable × Λ → Λ

\[
\begin{align*}
\text{sub′} (P, x, v y) &= \text{if } (x == y) \text{ then } P \text{ else } (v y) \\
\text{sub′} (P, x, M \cdot N) &= (\text{sub′}(P, x, M)) \cdot (\text{sub′}(P, x, N))
\end{align*}
\]

\[
\begin{align*}
y \neq x & \land \\Rightarrow \text{sub′} (P, x, \lambda y M) = \lambda y (\text{sub′}((v 0) \cdot M, x, M)) \\
y \neq 0 & \land y \neq f v(P) & \Rightarrow y \neq x \land y \neq 0 \land y \not\in f v(P)
\end{align*}
\]

To implement this function with our iterator principle we must change the order of the parameters, so our iterator principle now returns a function that is waiting for the term to be substituted. In this way we manage to vary the parameter through the iteration.

\[
\begin{align*}
hvar : & \text{Atom} \rightarrow \text{Atom} \rightarrow \Lambda \rightarrow \Lambda \\
hvar x y \text{ with } x ^= a y & \Rightarrow \\
\text{... } \text{yes } _= \text{id} & \Rightarrow \\
\text{... } \text{no } _= \lambda \_ \rightarrow (v y) & \Rightarrow \\
\text{sub′} : & \text{Atom} \rightarrow \Lambda \rightarrow \Lambda \rightarrow \Lambda \\
\text{sub′} x M P = & \Lambda t (\Lambda \rightarrow \Lambda) \\
& (hvar x) \\
& (f g N \rightarrow f N \cdot g N) \\
& (x :: 0 :: f v P, \lambda a f N \rightarrow \lambda a (f ((v 0) \cdot N))) \\
& M P
\end{align*}
\]

References


Defining the semantics of proof evidence

Dale Miller

Inria Saclay and LIX, École Polytechnique, France

Abstract

If automatic and interactive theorem provers store completed proofs, they do so using a range of proof structures, such as natural deduction, tableaux, resolution, and winning strategies. Ad hoc prover-specific proof scripts are also commonly used. I will outline how recent results on focused proof systems can be used to provide a formal framework for defining the meaning of a wide range of proof evidence. Interpreters of such formal definitions can thus be used as proof checkers. In order to make it possible to elide many details in formal proofs, proof checkers will be expected to perform significant proof reconstruction via deterministic and non-deterministic computations: such mixing of computation and deduction is an explicit feature of focused proof systems. I will discuss some of the ramifications of employing this framework on the ability of machines and humans to trust and communicate formal proofs.

Keywords: Proof certificates, focused proof systems.

References


---

1 Email: dale.miller@inria.fr

Selected, revised papers will be published in Electronic Notes in Theoretical Computer Science

URL: www.elsevier.nl/locate/entcs
Multi-focused Proofs with Different Polarity Assignments

Elaine Pimentel
DMAT, Universidade Federal do Rio Grande do Norte, Brazil.

Vivek Nigam
DI, Universidade Federal da Paraíba, Brazil.

João Neto
DMAT, Universidade Federal do Rio Grande do Norte, Brazil.

Abstract
In this work, we will reason on how a given focused proof where atoms are assigned with some polarity can be transformed into another focused proof where the polarity assignment to atoms is changed. This will allow, in principle, transforming a proof obtained using one proof system into a proof using another proof system. More specifically, using the intuitionistic focused system LJF restricted to Harrop formulas, we define a procedure, introducing cuts, for transforming a focused proof where an atom is assigned with positive polarity into another focused proof where the same atom is assigned negative polarity and vice-versa. Then we show how to eliminate these cuts, obtaining a very interesting result: while the process of eliminating a cut on a positive atom gives rise to a proof with one smaller cut, in the negative case the number of introduced cuts grows exponentially. We end the paper by showing how to use maximal multi-focusing identify proofs in LJF, giving rise to a 1-1 translation between maximal proofs in LJF and proofs in the natural deduction system for intuitionistic logic NJ, restricted to Harrop formulas.

Keywords: Intuitionistic logic, Proof Systems, Focusing, Identity of proofs.

1 Introduction
In focused proof systems, such as Andreoli’s original focused proof system [And92] for linear logic or Liang and Miller’s LJF and LKF focused proof systems for intuitionistic and classical logics [LM09], connectives are classified as positive or negative, according to their right introduction rules: positive connectives have not necessarily invertible rules, while negative connectives are those whose right introduction rules are invertible. The polarity of a non atomic formula is then given

1 Email: elaine.pimentel@gmail.com
2 Email: vivek.nigam@gmail.com
3 Email: joaoneto.ot@gmail.com
by the polarity of its outermost connective. The interesting fact is that atomic formulas can be arbitrarily assigned as positive or negative, without affecting the completeness of the focusing discipline.

While this choice for the polarity of atomic formulas does not affect provability, it does affect the shape of the resulting focused proofs obtained. For instance, in [NM10] it is shown that depending on the polarity assignments used for the atomic formula, one can, from the same logical theory, encode sequent calculus proofs or natural deduction ones. For another example, it has been shown in [CPP08] that this choice of polarities can explain different proof search strategies, such as backward chaining and forward chaining. More specifically, focusing and the polarity of atoms were used in order to justify proof theoretically the derivation steps used in the inverse method proof search mechanism. The results are over atoms in Horn theories only.

In this paper we consider a more general setting. In fact, using the focused system LJF [LM09] for intuitionistic logic restricted to hereditary Harrop formulas [MNPS91], we define a procedure, introducing cuts, for transforming a focused proof where an atom is assigned with positive polarity into another focused proof where the same atom is assigned negative polarity and vice-versa. We then show how to eliminate these cuts. Hence, we are able to transform a proof using a forward chaining strategy into a proof using backward chaining strategy or even obtain novel translations from sequent calculus to natural deduction and vice versa.

Interestingly, while the process of eliminating a cut on a positive atom gives rise to a proof with one smaller cut, in the negative case the number of introduced cuts grows exponentially. This difference in the cut-elimination algorithm is most definitely related to the different evaluation strategies according to the Curry-Howard isomorphism, where cut-elimination corresponds to computation in a functional programming setting (this is not addressed in this paper, though – we plan to investigate this better in the future).

We also propose a new multi-focused system for intuitionistic logic, mLJF, and show how to identify proofs in this system modulo permutations. It turns out that these maximal proofs, when restricted to Harrop formulas, give some very interesting results: if atoms are restricted to the negative polarity, mLJF collapses to LJF, while if atoms are restricted to the positive polarity, for each provable sequent in LJF there is exactly one maximal proof. This means that a proof with negative atoms correspond to a proof with positive atoms and the correspondence is 1-1 up to permutation of rules. In this way we are able, for the first time, to give a correspondence between an intuitionistic focused system with positive atoms and Gentzen’s natural deduction system NJ, solving completely the problem of identity of proofs in intuitionistic logic in the sequent calculus setting.

Finally, we sketch the dynamics of this correspondence in both sides, hence combining everything presented in the body of the paper.

The paper is organised as follows: Section 2 presents the system LJF and the logic programming fragment based on Harrop formulas, LJF$_H$; then Sections 3 and 4 show how to change polarities of atoms in LJF$_H$ (introducing cuts) and how to eliminate cuts coming back to proofs in LJF$_H$; Section 5 presents the multi-focused system mLJF and the notion of maximal multi-focused proofs; Section 6 relates
polarities with maximality in LIF and Section 7 concludes the paper and presents some ideas for continuing this work.

2 The focused proof system LIF for intuitionistic logic

There is a number of ways of defining a focused system from Gentzen’s sequent system LJ for intuitionistic logic [Gir93,Her94,DJS95,DL06,LM07,LM09]. We choose the one first presented in [LM07], called LIF since it is the only one which allows positive and negative atoms in the same system.

In order to present the focused proof system LIF, we first classify the connectives $\lor^+, \land, \exists, true$ and $false$ as positive (their left introduction is necessarily invertible) and the connectives $\supset, \land^-$, and $\forall$ as negative (their right introduction rules are invertible). This dichotomy must also be extended to formulas. Concerning the atomic ones: some pre-chosen atoms are considered negative and the rest are considered positive. That is, one is free to assign as positive or negative the polarity to atoms. From this, a formula is positive if its main connective is positive or it is a positive atom and is negative if its main connective is negative or it is a negative atom.

The proof system LIF depicted in Figure 1 has four types of sequents.

(i) The sequent $\Gamma; \cdot \rightarrow [R]$ is a right-focusing sequent (the focus is $R$);
(ii) The sequent $\Gamma, [R]; \cdot \rightarrow P_a$ is a left-focusing sequent (with focus on $R$);
(iii) The sequent $\Gamma; \Theta \Rightarrow R$ is an unfocused sequent. Here, $\Gamma$ contains negative formulas and positive atoms;

Fig. 1. The LJF system. Here $A_n$ denotes a negative atom, $P_a$ a positive atom, $P$ a positive formula, $N$ a negative formula, $P_a$ a positive formula or an atom and $\Omega$ is a multiset of negative or atomic formulas. All other formulas are arbitrary and $y$ is not free in $\Gamma, \Theta$ or $R$. 

71
(iv) The sequent $\Gamma; \vdash P_a$ is an instance of the previous sequent where $\Theta$ is empty and the formula in the succedent is positive or atomic.

As an inspection of the inference rules of LJF reveals, the search for a focused proof is composed of two alternating phases, and these phases are governed by polarities. The negative phase applies invertible (negative) rules until exhaustion: no backtracking during this phase of search is needed. The negative phase uses the third type of sequent above (the unfocused sequents): in that case, $\Theta$ contains positive or negative formulas, while $R$ is either negative or positive. If $\Theta$ contains positive formulas, then an introduction rule (either $\land$, $\exists$, $true_l$, or $false_l$) is used to decompose it; negative formulas are moved to the $\Gamma$ context (by using the store rule); if $R$ is negative, the rules $\land^-, \lor_r$ are applied until $R$ becomes positive or atomic. The end of the negative phase is represented by the fourth type of sequent. Such a sequent turns then to a focused one by using one of the decide rules, $D_r$ or $D_l$. The application of one of these decide rules then selects a formula for focusing and switches proof search to the positive phase or focused phase. This focused phase then proceeds by applying sequences of inference rules on focused formulas: in general, backtracking may be necessary in this phase of search. The focusing phase ends with one of the release rule $R_l$ or $R_r$.

As pointed out in [LM07], if all atoms are given negative polarity, the resulting proof system models backward chaining proof search and uniform proofs [MNPS91]. If positive atoms are permitted as well, then forward chaining steps can also be accommodated. Moreover, as in [NM10], it is possible in LJF to specify with the same intuitionistic theory sequent calculus proofs by using one polarity assignment and natural deduction proofs by using another polarity assignment.

**Example 2.1** It is well known that the polarity assigned to atomic formulas does not change provability. On the other hand, the shape of proofs can differ a lot when different polarities are assigned to atoms. Consider the Fibonacci program

$$\text{fib}(0,0) \land^+ \text{fib}(1,1) \land^+ \forall n, d, d'. [\text{fib}(n, d) \land^+ \text{fib}(n + 1, d') \supset \text{fib}(n + 2, d + d')]$$

Let $\Gamma = \text{fib}(0,0), \text{fib}(1,1), \forall n, d, d'. [\text{fib}(n, d) \land^+ \text{fib}(n + 1, d') \supset \text{fib}(n + 2, d + d')]$

If $\text{fib}$ has negative bias, then the only possible proof of $\Gamma \rightarrow \text{fib}(12,144)$ is

$$\begin{align*}
\pi_1 & : \text{fib}(0,0) \rightarrow \text{fib}(10,55) \\
\pi_2 & : \text{fib}(11,89) \rightarrow \text{fib}(10,55) \\
\pi_3 & : \text{fib}(12,144) \rightarrow \text{fib}(0,0) \\
\pi_4 & : \text{fib}(12,144) \rightarrow \text{fib}(1,1)
\end{align*}$$

where $\pi_1$ and $\pi_2$ continue following the backward chaining strategy. On the other hand, if $\text{fib}$ is positive, the only possible way to start the proof is the following

$$\begin{align*}
\pi_5 & : \text{fib}(0,0) \rightarrow \text{fib}(0,0) \\
\pi_6 & : \text{fib}(1,1) \rightarrow \text{fib}(0,0) \\
\pi_7 & : \forall n, d, d'. [\text{fib}(n, d) \land^+ \text{fib}(n + 1, d') \supset \text{fib}(n + 2, d + d')] \rightarrow \text{fib}(12,144)
\end{align*}$$
where $\pi_3$ can mix forward and backward chaining strategies. Note that the first derivation is exponential on size, while the smallest one in the second is linear.

The following result is trivially true, since right focused rules do not introduce left focused sequents.

**Lemma 2.2** Let $\Gamma$ be a set of LJF-formulas. Let $\Xi$ be a positive trunk, that is a derivation containing only rules from the positive phase, with end sequent of the form $\Gamma; \cdot \Rightarrow [F]$, then there is no sequent focused on the left in $\Xi$.

### 2.1 The logic programming fragment: $\text{LJF}_H$

In some parts of this paper (Sections 3, 4 and 6), we will restrict theories used to be the $D$-formulas and goals to be the $G$-formulas both specified by the grammar below

\[
G ::= A \mid G \land^+ G \mid D \supset G \mid \forall x G
\]

\[
D ::= A \mid G \supset A \mid \forall x.D
\]

That is, we will only consider, in those Sections, sequents of the type $D \vdash G$, where $D$ is a set of $D$-formulas and $G$ is a goal. This is a straightforward extension of the fragment of hereditary Harrop formulas used to describe uniform proofs [MNPS91], where $A$ is an atomic formula. We will call the resulting system $\text{LJF}_H$.

We restrict our language to this fragment mainly for presentation reasons, as it considerably simplifies the machinery used in the following sections. In particular, it allows for a concise cut-elimination procedure involving only some cut permutations shown in Section 4, which will be used in the subsequent sections to demonstrate the connections of the polarity assignment to translation of proofs in different systems, as well as giving a hint on how the change of polarities gives rise to call-by-value and call-by-name reduction strategies.

### 3 Changing polarities

In this section, we show how to transform focused proof where an atom is assigned with one polarity to a focused proof where this same atom is assigned the opposite polarity. For this, we may introduce inter-phase atomic cuts, that is, we may use the admissible cut rule in $\text{LJF}$

\[
\frac{\Gamma; \cdot \Rightarrow A\quad \Gamma, A; \cdot \Rightarrow C}{\Gamma; \cdot \Rightarrow C} \quad \text{cut}
\]

The transformations below might not preserve the size of a proof. In fact, it may well happen that after a proof is transformed from one proof system to another, the proof increases exponentially. Although this is relevant in some cases, such as in Proof Carrying Code, it is not that relevant when trying to unify the library of results obtained with different proof systems.

#### 3.1 From positive to negative polarity

In this section we demonstrate how to transform a focused proof where an atom is assigned with positive polarity into another focused proof where the same atom is
assigned negative polarity. Assume that Ξ is a proof where the atom A is assigned with positive polarity. We modify Ξ by induction from the leaves to the root on the number of reaction left and initial right rules applied on A. In particular, we perform the following operations:

The base case is when the proof ends with an initial right rule, which can only appear in positive derivations. We eliminate initial right rules by replacing the following subderivations appearing in a positive derivation:

\[
\begin{align*}
Γ;· &→ [A] \quad \text{I}_r \\
Γ; &⇒ A \quad \text{D}_r
\end{align*}
\]

by the following derivations, respectively:

\[
\begin{align*}
Γ, [A];· &→ A \quad \text{I}_l \\
Γ; &⇒ A \quad \text{D}_l \\
Γ;· &→ [A] \quad \text{R}_r \\
Γ; &⇒ [A] \quad \text{D}_l
\end{align*}
\]

Notice that from the former derivations, it is the case that A ∈ Γ and therefore we can, in the latter derivations, focus on A.

The other possible cases are when one of the rules ⊃_l, ∧_l or ∀_l are applied. In those cases, an instance of the cut rule is added. We illustrate the case of ⊃_l, the others are similar and simpler.

\[
\begin{align*}
Ξ_1 \quad Γ;A;· &⇒ G' \quad \text{R}_l, \text{store} \\
Γ;[G ⊃ A];· &⇒ G' \quad \text{D}_l \\
Γ; &⇒ G' \\
Ξ_2 \quad Γ; &⇒ A \\
\end{align*}
\]

Here, the derivations Ξ₁ and Ξ₂ are obtained by applying the inductive hypothesis to Ξ₁ and Ξ₂ of smaller height and transforming all occurrences of A with positive polarity into negative polarity. Notice that, from Lemma 2.2, in the remaining of positive trunk in Ξ₁ there may not be any occurrences of reaction left rules, but only of initial right rules which are handled by the base case. Hence, this operation removes all reaction left rules over all the appearances of the atomic formula A.

Finally, after applying these operations, we obtain an LJF proof with cuts. To obtain a cut-free proof, we apply the cut-elimination theorem given in Section 4. The resulting proof is a cut-free focused proof where the polarity of the atom A is negative.

### 3.2 From negative to positive polarity

The idea to transform a proof where an atom A is assigned with negative polarity to a proof where the same atom appears with positive polarity is similar to the previous case. We perform the following operations to the original proof:
To eliminate all occurrences of $R_r$, we will make use of the cut rule. Consider the following positive derivation containing $R_r$ rules on the negative polarity atom $A$ and whose last rule is $D_r$:

$$
\frac{\Xi_1 \quad \Xi_i \quad \Xi_n}{\Gamma; \vdash [G_1] \quad \cdots \quad \Gamma; \vdash [A] \quad \cdots \quad \Gamma; \vdash [G_n]} \quad \Gamma; \vdash [G] \quad \Gamma; \vdash G
$$

It can be transformed to the following derivation where $A$, where the number of reaction rules is reduced and this occurrence of $A$ has positive polarity.

$$
\frac{\Xi'_1 \quad \Xi'_i \quad \Xi'_n}{\Gamma; \vdash [G_1] \quad \cdots \quad \Gamma; \vdash [A] \quad \cdots \quad \Gamma; \vdash [G_n]} \quad \frac{\Xi'_1 \quad \Xi'_i \quad \Xi'_n}{\Gamma; \vdash [G] \quad \Gamma; \vdash G \quad \text{cut}} \quad \frac{\Xi'_1 \quad \Xi'_i \quad \Xi'_n}{\Gamma; \vdash A \quad \Gamma; \vdash G \quad \text{cut}}
$$

The proofs $\Xi'_1, \ldots, \Xi'_n$ are obtained by applying the inductive hypothesis where $A$ has positive polarity. The inductive hypothesis is applicable since their height are smaller and the number of reaction rules is decreased by at least one.

4 Cut-elimination

Instead of using the cut-elimination algorithm with several intra-phase cut-rules given in [LM09], we exploit the fact that the theories encoding proof systems are hereditary Harrop formulas to give a simpler cut-elimination procedure, with only inter-phase cut-rules.

4.1 If the cut-formula is a positive atom

Our algorithm consists of basically two rewrite rules, depending on which decide rule is applied last on left premise of the cut rule. If it is $D_r$ then it is necessarily the case that the atom $A$ used in the cut is in the context $\Gamma$, which implies that the cut is not necessary:
This derivation reduces to the following derivation where the cut is eliminated:

\[
\Xi \\
\Gamma; \cdot \Rightarrow G
\]

For the second case, when the decide rule \(D_l\) is applied last in the left premise of the cut rule, we proceed as follows:

\[
\begin{array}{c}
\Xi_1 \\
\Gamma, A'; A \Rightarrow R_l, \text{store} \\
\Xi_2 \\
\Gamma, A; \cdot \Rightarrow G \quad \text{cut}
\end{array}
\]

\[
\begin{array}{c}
\Gamma_1; \cdot \Rightarrow [B_1] \\
\Gamma_n; \cdot \Rightarrow [B_n] \\
\Gamma, [A']; \cdot \Rightarrow A \\
\Gamma; \cdot \Rightarrow A \\
\Gamma, [A']; \cdot \Rightarrow A \\
\Gamma; \cdot \Rightarrow G \\
\end{array}
\]

Since our theories are hereditary Harrop formulas, once the formula \(F\) is focused on, the resulting formula focused on the left is necessarily an atom. Moreover, the atom \(A'\) cannot be negative otherwise one would have to finish the proof with an \(I_l\) rule, but this is not possible since the atom appearing at the right-hand-side, \(A\), is positive. Hence, it is necessarily the case that the atom \(A'\) is positive and since it is focused on the left, one releases focus.

We permute the atomic cut above the positive phase to the left as follows:

\[
\begin{array}{c}
\Xi_1 \\
\Gamma, A'; A \Rightarrow G \\
\Xi_2 \\
\Gamma, A; \cdot \Rightarrow G \quad \text{cut}
\end{array}
\]

\[
\begin{array}{c}
\Gamma_1; \cdot \Rightarrow [B_1] \\
\Gamma_n; \cdot \Rightarrow [B_n] \\
\Gamma, [A']; \cdot \Rightarrow G \\
\Gamma; \cdot \Rightarrow G \\
\end{array}
\]

**Remark 4.1** Observe that the cut is replaced by another, appearing upper in the proof.

### 4.2 If the cut-formula is a negative atom

It turns out that the cut may not permute upwards on the left premise if \(A\) is negative. In fact, on focusing on a left formula \(F\) like in the last Section, if the resulting atom focusing on the left is negative, it has necessarily to be \(A\) and the proof finishes with an \(I_l\) rule. For all other cases we could proceed like in the positive case.

There are two base cases:

\[
\begin{array}{c}
\Xi \\
\Gamma; \cdot \Rightarrow A \\
\Xi \\
\Gamma, A'; A \Rightarrow A' \\
\Xi \\
\Gamma, A'; A' \Rightarrow A \\
\end{array}
\]

\[
\begin{array}{c}
\Xi \\
\Gamma; \cdot \Rightarrow A \\
\Xi \\
\Gamma, A'; A \Rightarrow A' \\
\Xi \\
\Gamma, A'; A' \Rightarrow A \\
\end{array}
\]
The inductive cases are obtained by moving the cut rule upwards.

Let ⋆ be the maximum sequence of inference rules excluding decide rules appearing above the sequent \( \Gamma, A ; \vdash G \) (hence ⋆ has only negative rules). Let \( n \) be the minimum length of the sub-derivations of ⋆. If \( n > 0 \),

\[
\begin{align*}
\Xi & \vdash A \\
\Gamma; \vdash A; \vdash G & \quad \text{cut} \\
\Gamma; \vdash G & \quad \ast
\end{align*}
\]

where \( \Gamma \subseteq \Gamma' \).

If, on the other hand, \( n = 0 \), the last rule applied for proving \( [\Gamma, A] \rightarrow [G] \) is a decision rule. There are then two sub-cases: \( D_l \) and \( D_r \).

In both cases, after finishing the focus phases (positive or negative) we will end up with a proof of the shape (ignoring the leaves):

\[
\begin{align*}
\Xi & \vdash A \\
\Gamma_1; \vdash G_1 & \quad \text{cut} \\
\Gamma_n; \vdash G_n & \\
\Gamma; \vdash G & \quad \ast
\end{align*}
\]

and the cut is moved upwards as follows:

\[
\begin{align*}
\Xi & \vdash A \\
\Gamma_1; \vdash G_1 & \quad \ast \\
\Gamma_n; \vdash G_n & \\
\Gamma; \vdash G & \quad \ast
\end{align*}
\]

Remark 4.2 Observe that, in this case, one cut is replaced by many others, and hence the size of proof grows exponentially.

5 Multi-focusing

It is well known [Her94,EDH15] that the negative fragment of sequent calculus corresponds to natural deduction proofs. For example, the sequent \( a, a \supset b, b \supset c \vdash c \) has two different proofs in LJ:

\[
\begin{align*}
\text{forward:} & \quad a \vdash a & \quad & b \vdash b & \quad & c \vdash c & \quad & \Gamma \\
& \quad \quad \quad I & \quad & \quad I & \quad & \quad I & \quad & \Gamma \\
\text{backward:} & \quad a, a \supset b & \vdash & b, b \supset c & \vdash & c & \vdash & \Gamma
\end{align*}
\]

The first proof corresponds to forward and the second backward chaining. In LJF, if atoms are positive the only proof is the first one, while if they are negative, the only valid proof is the second.

In natural deduction, there is only one proof:

\[
\begin{align*}
\text{forward:} & \quad a, a \supset b, b \supset c \vdash c & \quad & a, a \supset b, b \supset c \vdash a & \quad & a, a \supset b, b \supset c \vdash b & \quad & a, a \supset b, b \supset c \vdash E \\
& \quad \quad \quad I & \quad & \quad I & \quad & \quad I & \quad & \Gamma
\end{align*}
\]
which corresponds to the negative proof.

Example 5.1 Consider the sequent $\Gamma; \cdot \Rightarrow b \land^+ d$ where $\Gamma = \{a, c, a \supset b, c \supset d\}$.
This sequent has 6 different proofs in LJF. If all atoms are negative, the only possible proof is

$$
\Gamma; \cdot \Rightarrow b \land^+ d
$$

But if atoms are positive, there are two possible proofs:

$$
\Gamma; \cdot \Rightarrow b \land^+ d
$$

and

$$
\Gamma; \cdot \Rightarrow b \land^+ d
$$

Observe that the proofs differ only in the order of the application of the implication.

We will show next how to use the maximal multi-focusing approach in order to identify proofs that differ only on the permutation of rules. We start by presenting mLJF, a multi-focused system for LJF.

The system mLJF has two kinds of formulas:

$$
P, Q ::= A_p \mid false \mid true \mid P \land^+ Q \mid P \lor Q \mid \exists x. P(x) \mid \downarrow N
$$

$$
M, N ::= A_n \mid M \land^- N \mid P \supset N \mid \forall x. N(x) \mid \uparrow P
$$

where $P, Q$ are positive while $M, N$ are negative formulas. The symbols $\uparrow$ and $\downarrow$ mark the changing of polarities. The syntax for contexts is the following

$$
\Delta ::= \cdot \mid \Delta, N \quad \Gamma, \Omega ::= \Delta \mid p \quad \Psi ::= [\Delta] \quad \Theta ::= \cdot \mid \Theta, P - \{p\}
$$

Finally, mLJF has three kinds of sequents:
Pimentel, Nigam, and Neto

Positive Phase

\[ \Gamma; \rightarrow [true] \quad \Gamma, \Psi; \rightarrow [B \land^+ C] \quad \Gamma, \Psi; \rightarrow [B \lor B_2] \quad \Gamma, [B_1 \lor B_2]; \rightarrow R \]

Structural Rules

\[ \Delta, \Gamma, \Theta; \rightarrow P_a \quad mD_l \quad \Delta, \Gamma, \Theta; \rightarrow \uparrow P \quad mD_r \quad \Gamma; \Theta \Rightarrow \downarrow N \quad mR_r \]

Fig. 2. mLJF system. Here \( A_n, A_p, P \) and \( N \) are the same as in Figure 1, \( P_a \) represents either a formula of the kind \( \uparrow P \) or an atomic formula and \( R \) is either \( P_a \) or a bracket formula. In \( mD_l \), \( \Delta \) is non empty.

- the sequent \( \Gamma; \Theta \Rightarrow R \) is unfocused;
- the sequent \( \Gamma, \Psi; \rightarrow R \) is focused on the left, where \( \Psi \neq \emptyset \);
- the sequent \( \Gamma, \Psi; \rightarrow [R] \) is focused on the right (and possibly on the left).

The negative phase in mLJF is the same as in LJF. The rest of the rules for mLJF are similar to the ones presented in Figure 1, only now considering possibly multi-focused contexts (Figure 2). Note that we can unfocus if and only if every focused formula is marked with arrows.

The following theorem is straightforward: just note that if we erase the \( \uparrow \) and \( \downarrow \) arrows and the context \( \Psi \), and if we restrict \( \Delta \) to a singleton in \( mD_l \) and to the empty set in \( mD_r \), mLJF collapses to LJF.

**Theorem 5.2** mLJF is correct and complete with respect to LJF.

Observe that the rule \( \supset \) has a “linear” flavour as the focused left context splits on the premise sequents. This is only an operational trick in order to make multi-localization possible.

**Example 5.3** If restricted to positive atoms, there are now four proofs of the sequent presented in Example 5.1: focusing on \( a \supset \uparrow b \) first, focusing on \( c \supset \uparrow d \) first, or focusing on both at the same time and then applying the implication rules in the two possible orders. These two last proofs collapse to one if we consider the equivalent class of proofs modulo permutation of rules

\[ \Gamma; \rightarrow [a] \quad \Gamma; \rightarrow [c] \quad \Gamma, b, d; \rightarrow [b \land^+ d] \quad mR_r \]

\[ \Gamma, [a \supset \uparrow b, c \supset \uparrow d]; \rightarrow b \land^+ d \quad mD_l \quad 2 \times (\supset) \]

79
In this case, we say that the application of $mD_l$ rule is maximal, that is, if it chooses the maximal possible set $\Delta$ for focusing. And it gives rise to a synthetic connective [MP13], that is, a connective that combines the application of various rules in one. Finally, observe that this maximal proof is possible only due to the splitting of the left focused context in the rule $\supset_l$, since the application of $I_r$ on proving $a$ and $c$ implies that we cannot have any other focused formulas.

### 5.1 Maximal multi-focusing

We will now formalise the notion of maximal multi-focusing and equivalence of proofs, presented intuitively in the last example.

The following definitions are adaptations of the ones in [CMS08,CHM12] to mLJF:

**Definition 5.4** The proofs $\Xi_1$ and $\Xi_2$ of the same mLJF sequent are locally permutatively equivalent, written $\Xi_1 \sim \Xi_2$, if each can be rewritten to the other using local permutations. $\Xi_1$ and $\Xi_2$ are permutatively equivalent, written $\Xi_1 \approx \Xi_2$, if they are locally permutatively equivalent and each can be rewritten to the other using permutations.

For example,

\[
\begin{align*}
\Gamma; \Theta, B, C, D \Rightarrow E & \quad \sim \quad \Gamma; \Theta, B, C \Rightarrow D \supset E \quad \supset_r^+ \\
\Gamma; \Theta, B \wedge^+ C \Rightarrow D \supset E & \quad \sim \quad \Gamma; \Theta, B \wedge^+ C \Rightarrow D \supset E \quad \supset_r^+
\end{align*}
\]

In fact, since all negative rules are invertible, they are permutable. This means that the whole negative phase collapse to one step, modulo permutations.

In the positive phase the permutability of rules depends on the polarities of formulas. We will come back to this later.

Non-locally permutatively equivalent proofs, on the other hand, require considering permutations of entire phases. As in [And01,CMS08], we call a neighbouring pair of phases, with the bottom phase positive and the top phase negative, a bipole. Consider two neighbouring bipoles: if the positive phase of the top bipole permutes with the negative phase of the bottom bipole, then in an unfocused form we can perform the permutation and merge the two bipoles by uniting their positive and negative phases, obtaining another (multi-)focused proof. This operation obviously terminates, giving rise to the following definition and theorem.

**Definition 5.5** If a proof $\Xi$ in mLJF ends with an instance of $mD_l$ or $mD_r$, let $\text{foci}(\Xi)$ is defined as the multiset of foci in the premise of that instance. We say that this instance of $mD_l$ or $mD_r$ is maximal if and only if, for every $\Xi' \approx \Xi$, $\text{foci}(\Xi') \subseteq \text{foci}(\Xi)$. A proof in mLJF is maximal if and only if every instance of $mD_l$ or $mD_r$ in it is maximal.

**Theorem 5.6** Every sequent provable in mLJF has a maximal proof.

The proofs presented in Example 5.3 are maximal, while the last two ones in Example 5.1 are not. But they can be transformed, via non-local permutations, to the ones in Example 5.3.
6 Maximal multi-focusing and Harrop formulas

The restriction of mLJF to Harrop formulas (here called mLJFH) gives very interesting results.

**Theorem 6.1** If all atoms are negative then mLJFH = LJFH, that is, when restricted to Harrop formulas, multi-focused proofs are the same as singly focused proofs if only negative atoms are considered.

**Proof.** Consider the proof

\[
\frac{\Xi_1 \quad \Xi_2}{\Gamma, G \supset A, \Psi_1: \rightarrow [G] \quad \Gamma, G \supset A, \Psi_2, [A]: \rightarrow C} \quad \supset_l \quad \frac{\Gamma, G \supset A, \Psi_2, [G \supset A]: \rightarrow C}{\Gamma, G \supset A: \Rightarrow C} \quad mD_l
\]

If \(A\) is a negative atom, \(\Xi_2\) must be the application of the initial axiom \(I_l\) and hence \(A = C\) and \(\Psi_2 = \emptyset\). Now, it should be the case that \(\Psi_1 = \emptyset\). If not, observe that it cannot exist a negative atom \(n \in \Psi_1\), since \(G\) is focused on the right (and focused negative atoms should finish the proof). Hence either there exists \(G' \supset A'\) or \(\forall x. D\) in \(\Psi_1\). But applying \(\supset_l\) in a sequent of the type \(\Gamma, G \supset A, \Psi_1: \rightarrow [G]\) will produce a sequent of the form \(\Gamma, G \supset A, \Psi'_1, [A']: \rightarrow [G]\), which is forbidden since \(A'\) is atomic negative (hence there can be no focused formula on the right of the sequent). On the other hand, applying \(\forall l\) will substitute a focused formula \(\forall x. D\) by the focused formula \(D\); in this case, the focused context on the left will always produce another one, and the result follows by induction.

That is, there are no non-local permutations, foci in maximal multi-focused formulas have exactly one element, hence mLJFH = LJFH. The other cases are similar and simpler.

**Corollary 6.2** If all atomic formulas are negative, any provable sequent in mLJFH has only one possible proof.

In the positive case we also have a fascinating result.

**Theorem 6.3** For each provable sequent in mLJFH, if all atoms are positive then there is only one maximal proof for it. That is, when restricted to Harrop formulas with only positive atoms, multi-focused proofs can be equated to one maximally focused proof.

**Proof.** Consider the maximal proof \(\Xi\)

\[
\frac{\Xi_1 \quad \Xi_2}{\Gamma, G \supset A, \Psi_1: \rightarrow [G] \quad \Gamma, G \supset A, \Psi_2, [A]: \rightarrow C} \quad \supset_l \quad \frac{\Gamma, G \supset A, \Psi_2, [G \supset A]: \rightarrow C}{\Gamma, G \supset A: \Rightarrow C} \quad mD_l
\]

If \(G\) is a purely positive formula, \(\Psi_1\) should be empty and there are no rules up to permute with the rightmost premise. If \(G = \downarrow N\), a number of things can happen:

1. If \(\Psi_1\) is a (possibly empty) set of the form \(\uparrow \Delta\), then focus will be lost and there
will be a change of phases. Since \( \Xi \) is maximal, there is no way of permuting these phases. If \( \Xi_1 \) ends with \( \forall_i \) or \( \exists_i \), then these rules are locally permutable with \( \forall_i \).

For example, if \( G' \supset A' \in \Psi_1 \) then

\[
\frac{\Xi' \quad \Xi''}{\Gamma, G \supset A; \Psi_1; \vdash [G']} \quad \frac{\Xi''}{\Gamma, G \supset A; \Psi_2, \vdash [A'] \\ \vdash [G]} \quad \Xi_2 \\
\frac{\Gamma, G \supset A; \Psi_2, \vdash [A'] \\ \vdash [G]} \quad \Xi_2 \\
\frac{\Gamma, G \supset A; \Psi_1, \Psi_2, \vdash [G \supset A]; \cdot \to C}{\Delta I} \quad mD_i
\]

is locally equivalent to

\[
\frac{\Xi' \quad \Xi''}{\Gamma, G \supset A; \Psi_1; \vdash [G']} \quad \frac{\Xi''}{\Gamma, G \supset A; \Psi_2, \vdash [A'] \\ \vdash [G]} \quad \Xi_2 \\
\frac{\Gamma, G \supset A; \Psi_2, \vdash [G \supset A', G' \supset A']; \cdot \to C}{\Delta I} \quad mD_i
\]

The analysis is similar and simpler for \( \Psi_2 \) or in the case that multi-focusing is also on the right (\( mD_r \)).

**Corollary 6.4** There is a 1-1 correspondence between maximal proofs in mLJF\(_H\) restricted to positive atoms and proofs in mLJF\(_H\) restricted to negative atoms. Hence there is a 1-1 correspondence between mLJF\(_H\) restricted to positive atoms and proofs in NJ restricted to Harrop formulas.

We will finish this section by sketching how these correspondences work, using the process developed in Sections 3 and 4.

**From positive to negative.** The process of changing polarities of atoms will transform a cut-free proof in mLJF\(_H\) into a proof with cuts.

\[
\frac{\Xi_1 \quad \Xi_2}{\Gamma; \cdot \to [G']} \quad R_l; \text{store} \quad \frac{\Xi_1 \quad \Xi_2}{\Gamma; \cdot \to [G']} \\
\frac{\Gamma; \cdot \to [G]}{\Gamma; \cdot \to G'} \quad D_l
\]

We will denote by \( \Xi \) the leftmost subproof above the cut.

The cut-elimination process on negative atoms will (i) permute down the focused rule on the right premise above the cut (if any) and (ii) add a higher cut to every possible top premise appearing when the focused phase is over\(^4\).

\[
\frac{\Xi_1}{\Gamma_1; \cdot \to A} \quad \frac{\Xi_1}{\Gamma_1; A_i; \cdot \to G_1} \quad \text{cut} \quad \frac{\Xi_1}{\Gamma_n; \cdot \to A} \quad \frac{\Xi_n}{\Gamma_n; A_i; \cdot \to G_n} \quad \text{cut}
\]

Consider the proof

\[
\frac{\Xi_i}{\Gamma_i; \cdot \to A} \quad \frac{\Xi_i}{\Gamma_i, A_i; \cdot \to G_i} \quad \text{cut}
\]

\(^4\) Here we abuse the notation and use \( \Xi \) also for its weakened version, substituting \( \Gamma \) by \( \Gamma_i \), where \( \Gamma \subseteq \Gamma_i \).
If the last rule of $\Xi_i$ is the identity on $A$, then $G_i = A$ and hence the proof above is substituted by $\Xi$. If the last rule of $\Xi_i$ is the identity on a formula other than $A$, then the cut is eliminated. Finally, if the last rule of $\Xi_i$ is not the identity, we continue moving the cut up, together with $\Xi$. This will eliminate all the uppermost cuts and completely determine the order of application of rules in the negative case.

As an example, if we take either of the last two proofs in Example 5.1, this process will give the first proof, where the conjunction moves down and the implications occur in parallel branches of the proof.

**From negative to positive.** The proof

\[
\begin{array}{c}
\Xi_i \\
\Gamma; \cdot \rightarrow [G_1] \\
\cdots \\
\Gamma; \cdot \rightarrow [A] \\
\cdots \\
\Gamma; \cdot \rightarrow [G_n] \\
\Xi_n \\
\end{array}
\]

is transformed into

\[
\begin{array}{c}
\Xi_i' \\
\Gamma, A; \cdot \rightarrow [G_1] \\
\cdots \\
\Gamma, A; \cdot \rightarrow [A] \\
\cdots \\
\Gamma, A; \cdot \rightarrow [G_n] \\
\Xi_n' \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma; \cdot \Rightarrow A \\
\Gamma, A; \cdot \Rightarrow G \\
\end{array}
\]

cut

We will call $\Pi_1$ the rightmost subproof above the cut. Now if $\Xi_i'$ has the form

\[
\begin{array}{c}
\Pi_1 \\
\Gamma_1; \cdot \rightarrow [B_1] \\
\cdots \\
\Gamma_n; \cdot \rightarrow [B_n] \\
\Gamma, A'; \cdot \Rightarrow A \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma, A'; \cdot \Rightarrow G \\
\Gamma, A'; \cdot \Rightarrow A \\
\end{array}
\]

we can move the cut up

\[
\begin{array}{c}
\Pi_2 \\
\Gamma, A'; \cdot \Rightarrow G \\
\Gamma, A'; \cdot \Rightarrow A \\
\end{array}
\]

cut

\[
\begin{array}{c}
\Pi_1 \\
\Gamma, A'; \cdot \Rightarrow G \\
\Gamma, A'; \cdot \Rightarrow A \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma_1; \cdot \rightarrow [B_1] \\
\cdots \\
\Gamma_n; \cdot \rightarrow [B_n] \\
\Gamma, A'; \cdot \Rightarrow G \\
\end{array}
\]

cut

\[
\begin{array}{c}
\Pi_2 \\
\Gamma_1; \cdot \rightarrow [B_1] \\
\cdots \\
\Gamma_n; \cdot \rightarrow [B_n] \\
\Gamma, A'; \cdot \Rightarrow G \\
\end{array}
\]

Observe that focusing on the right is eliminated and, depending on the choice of $A$ in $\Xi_i$, we may have different but permutatively equivalent proofs. In Example 5.1, starting from the first proof, we get the second proof if $A = a$ and the third if $A = c$.

### 7 Conclusion and future work

In this work, we have proposed a multi-focused system mLJF for the focused intuitionistic system LJF [LM07]. We then showed how to use the notion of maximal proofs in order to identify proofs in intuitionistic logic. The same results have been
established in [CMS08] for the multiplicative-additive fragment of linear logic and in [CHM12] for classical logic.

This is an important step towards solving the problem of identity of proofs in intuitionistic logic in the sequent calculus setting. In fact, when restricted to Harrop formulas, we have completely solved the problem (see Theorems 6.1 and 6.3). We hope to be able to expand these results for the whole intuitionistic logic.

But a very nice line of research to pursue is to relate the procedure given in Sections 3 and 4 in order to relate call-by-name and call-by-value. In particular, as noted in Remarks 4.1 and 4.2, systems restricted to positive atoms have a call-by-value behavior, where one cut is substituted by another on eliminating the cut. This has the flavour of linear reduction steps, evaluating the argument first for then passing it as a parameter. On the other hand, systems restricted to negative atoms have a call-by-name behavior, where one cut is substituted by possible many others, capturing well the notion of first passing the argument, then reducing all possible occurrences of it in the term.

References


84
On Strong Normalization in Proof-graphs for Propositional Logic

Marcela Quispe-Cruz
Universidad Catolica San Pablo, Peru
Edward Hermann Haeusler
Departamento de Informatica
Pontificia Universidade de Rio de Janeiro
Rio de Janeiro, Brazil
Lew Gordeev
Tuebingen University, Ghent University

Abstract

Traditional proof theory of Propositional Logic deals with proofs whose size can be huge. Proof theoretical studies discovered exponential gaps between normal or cut free proofs and their respective non-normal proofs. The use of proof-graphs, instead of trees or lists, for representing proofs is getting popular among proof-theoreticians. Proof-graphs serve as a way to provide a better symmetry to the semantics of proofs and a way to study complexity of propositional proofs and to provide more efficient theorem provers, concerning size of propositional proofs.

Mimp-graphs were initially developed for minimal implicational logic representing proofs through references rather than copy. Thus, formulas and sub-deductions preserved in the graph structure, can be shared deleting unnecessary sub-deductions resulting in the reduced proof. In this work, we consider full minimal propositional logic and show how to reduce (eliminating maximal formulas) these representations such that strong normalization theorem can be proved by simply counting the number of maximal formulas in the original derivation. In proof-graphs, the main reason for obtaining the strong normalization property using such simple complexity measure is a direct consequence of the fact that each formula occurs only once in the proof-graph and the case of the hidden maximum formula that usually occurs in the tree-form derivation is already represented in the mimp-graph.

Keywords: Proof Theory, Proof Graphs, N-Graphs, Intuitionistic Logic, Sequent Calculus, Multiple-Conclusion Systems.

Email: mquispec@ucsp.edu.pe
Email: hermann@inf.puc-rio.br
Email: lew.gordeev@uni-tuebingen.de

Selected, revised papers will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
1 Introduction

Recently the use of graphs instead of trees to represent proofs has been shown to be more efficient\[2\][1], while also being helpful to better address the lack of symmetry in classical ND logic \[3\] and to address the complexity of the proof normalization process. Previously we have already presented mimp-graphs as a new proof system developed for minimal implicational logic \[5\], whose deductions are structured as proof-graph. The point is that in mimp-graphs it is easy to determine maximal formulas\(^4\) and upper bounds on the length of reduction sequences leading to normal proofs.

Thus a normalization theorem is proved by counting the number of maximal formulas in the original derivation. The strong normalization property is a direct consequence of such normalization, since any reduction decreases the corresponding measure of derivation complexity. In the present paper we wish to explain this procedure more clearly and expand it onto full propositional logic.

Mimp-graphs are directed graphs whose nodes and edges are assigned with labels. Moreover we distinguish two parts, one representing the inferences of a proof, and the other the formulas. For the formula-part of a mimp-graph, we use formula dags that consist of basis in the mimp-graph construction (see Definition 2.2) and contain only formula nodes sharing formula nodes, thus each formula node only need to occur once in the graph, an example is shown in the Figure 1: the propositions \(P\) and \(Q\) occur once in the graph.

For the inference-part of a mimp-graph we have the rule nodes (R-nodes) that are labeled by the names of the inference rules. The logic connectives and inference names may be indexed, in order to achieve a 1-1 correspondence between formulas (inferences) and their representations (names).

We show in Figure 2 an example of the eliminating a maximal formula in a derivation in Natural Deduction. The formula \(P \rightarrow (P \land Q)\) is not a maximal formula before a reduction \((\land_1)\) is applied to eliminate the maximal formula \((P \rightarrow Q) \rightarrow (P \rightarrow (P \land Q)))\). This possibility of having hidden maximal formulas in ND is the main reason to use more sophisticated methods whenever proving strong normalization.

In Figure 3 we show an embedding of this derivation into a mimp-graph. This example shows the reason why our normalization procedure is strongly normalizing. We remark that there is no possibility to hide a maximal formula because all formulas are represented only once in the graph (see Figure 3). In this graph \(P \rightarrow (P \land Q)\) is already a maximal formula. We can choose to remove any of the maximal

\(^4\) A maximal formula is a formula occurrence that is consequence of a introduction rule and the major premise of a elimination rule.
formulas. If $P \rightarrow (P \wedge Q)$ is chosen to be eliminated, by the mimp-graph elimination procedure, its reduction eliminates the $(P \rightarrow Q) \rightarrow (P \rightarrow (P \wedge Q))$ too. On the other hand, the choice of $(P \rightarrow Q) \rightarrow (P \rightarrow (P \wedge Q))$ to be reduced only eliminates itself. In any case the number of maximal formulas decreases and the mimp-graph becomes as shown in Figure 4.

Fig. 2. Derivation of \((R, R \rightarrow P, R \rightarrow (P \rightarrow Q)) \vdash P \wedge Q\) with two steps of reduction.

Fig. 3. Mimp-graph translation of the example in Figure 2.

Fig. 4. Normalized mimp-graph of the example in Figure 3.
2 Propositional mimp-graphs

In [5] we considered implication as the only logic connective. Let us now turn to a more general presentation of proof-graphs for propositional calculus that includes implication, conjunction and disjunction that we called mimp-graph for propositional logic. First, we define sets of labels to nodes and edges of the graph. It will be defined along with partial ordering on its R-nodes that allows to pass through the nodes of the structure. We will also develop the normalization procedure for these proof-graphs.

Definition 2.1 [Label types] There are five types of labels:

- **R-Labels** is the set of labels for rule nodes: \( \{ \to E_m \}_{m \in \mathbb{N}} \cup \{ \to I_n \}_{n \in \mathbb{N}} \cup \{ \wedge I_0 \}_{o \in \mathbb{N}} \cup \{ \vee E_p \}_{p \in \mathbb{N}} \cup \{ \forall T_q \}_{q \in \mathbb{N}} \cup \{ \forall E_r \}_{r \in \mathbb{N}} \).
- **F-Labels** is the set of labels for formula nodes: \( \{ \to i, \wedge j, \vee k, i, j, k \in \mathbb{N} \} \) and the propositional letters \( \{ P, Q, R, \ldots \} \).
- **D-Labels** is the set of labels for delimiter nodes: \( \{ H_k | k \in \mathbb{N} \} \cup \{ C \} \).
- **E_F-Labels** is the set of labels for formula edges: \( \{ l \text{ (left)}, r \text{ (right)} \} \).
- **E_M-Labels** is the set of labels for rule edges: \( \{ p_i \text{ (premise)}, r p_j \text{ (right premise)}, l p_k \text{ (left premise)}, r m_l \text{ (right minor premise)}, l m_m \text{ (left minor premise)}, m_n \text{ (minor premise)}, M_o \text{ (major premise)}, c_p \text{ (conclusion)}, d i s c_q \text{ (discharge)}, h y p_r \text{ (hypothesis)}, c o n c \text{ (final conclusion)} / i, j, k, l, m, n, o, p, q, r \in \mathbb{N} \} \).

The union of these four sets of label types will be called LBL.

We will use the terms \( \alpha_m, \beta_n \) and \( \gamma_r \) to represent the principal connective of the formula \( \alpha, \beta \) and \( \gamma \) respectively.

Definition 2.2 A mimp-graph for propositional logic \( G \) is a directed graph \( \{ V, E, L \} \) where: \( V \) is a set of nodes, \( L \) is a set of labels, \( E \) is a set of labelled edges \( \{ v \in V, t \in L, v' \in V \} \), of source \( v \), of target \( v' \) and label \( t \).

The propositional mimp-graphs are recursively defined as follows:

**Basis** If \( G_1 \) is a formula graph with root node \( \alpha_m \) then the graph \( G_2 \) defined as \( G_1 \) with delimiter nodes \( H_n \) and \( C \) and edges \( \alpha_m \xrightarrow{c o n c} C \) and \( H_n \xrightarrow{h y p} \alpha_m \) is a mimp-graph.

**\( \to E \)** If \( G_1 \) and \( G_2 \) are mimp-graphs, and the graph (intermediate step) obtained by \( G_1 \oplus G_2 \) contains the edge \( \to \gamma l \alpha_m \) and two nodes \( \to \gamma q \alpha_m \) linked to the delimiter node \( C \), then the graph \( G_3 \) that is defined as \( G_1 \oplus G_2 \) with

(i) the removal of ingoing edges in the node \( C \) which were generated in the intermediate step (see the figure below, dotted area in \( G_1 \oplus G_2 \));

(ii) an R-node \( \to E_i \) at the top position;

(iii) the edges: \( \alpha_m \xrightarrow{m \text{ new}} E_i, \to \gamma q \xrightarrow{M \text{ new}} E_i, \xrightarrow{E_i e \text{ new}} \beta_n \) and \( \beta_n \xrightarrow{c o n c} C \), where \( \text{ new} \) is a fresh (new) index ranging over all edges of kind \( c, m \) and \( M \) ingoing and/or outgoing of the formula-nodes \( \alpha_m, \beta_n \) and \( \to \gamma q \);

is a mimp-graph (see figure below).
If \( G_1 \) is a mimp-graph and contains a node \( \beta_n \) linked to the delimiter node \( C \) and the node \( \alpha_m \) linked to the delimiter node \( H_k \), then the graph \( G \) that is defined as

(i) \( G := G_1 \oplus G_2 \), such that \( G_2 \) is a formula graph with root node \( \to_t \) linked to F-nodes \( \alpha_m \) and \( \beta_n \) by the edges: \( \to_l \alpha_m, \to_r \beta_n \);

(ii) with the removal of the edges: \( \beta_n \to_C \);

(iii) an R-node \( \to_I \) at the top position;

(iv) the edges: \( \beta_n \to_{\text{new}} \to_I, \to_I \to_C, \to_C \to_I \), where \( \text{new} \) is a fresh (new) index considering all edges of kind \( p, \text{disc} \) and \( c \) ingoing and/or outgoing of the formula-nodes \( \alpha_m, \beta_n \) and \( \to_y \);

is a mimp-graph (see figure below; the \( \alpha_m \)-node is discharged).

If \( G_1 \) and \( G_2 \) are propositional mimp-graphs and \( G_1 \) contains \( \alpha_m \) linked to the D-node \( C \) and \( G_2 \) contains \( \beta_n \) linked to the D-node \( C \), then the graph \( G \) that is defined as

(i) \( G := G_1 \oplus G_2 \oplus G_3 \) with the removal of the ingoing edges in the node \( C \) which were generated in the intermediate step (see figure below, dotted area in \( G_1 \oplus G_2 \oplus G_3 \));

(ii) an R-node \( \land_1 \) at the top position;

(iii) the edges: \( \alpha_m \to_{\text{new}} \land_1, \beta_n \to_{\text{new}} \land_1, \land_1 \to_C, \land_1 \to_C \), and \( \to_C \to_C \),

is a mimp-graph, see figure below.
If \( G_1, G_2 \) and \( G_3 \) are propositional mimp-graphs, and the graph obtained by \( (G_1 \odot G_2) \oplus G_3 \) (intermediate step) contains the nodes: \( \lor \) and \( \sigma_r \) linked to the D-node \( C \) (\( \sigma_r \) twice); and \( \alpha_m \) and \( \beta_n \) are subformulas of \( \lor \) and are linked to D-nodes \( H \), then the graph \( G \) that is defined as \( (G_1 \odot G_2) \oplus G_3 \) with

(i) the removal of the ingoing edges in the node \( C \) which were generated in the intermediate step (see figure below);
(ii) an R-node \( \lor \) at the top position;
(iii) the edges: \( \sigma_r \xrightarrow{\text{lmw}} \lor \), \( \sigma_r \xrightarrow{\text{rmw}} \lor \), \( \lor \xrightarrow{\text{Mw}} \lor \), \( \lor \xrightarrow{\text{cw}} \sigma_r \), \( \lor \xrightarrow{\text{idiscw}} H_u \), \( \lor \xrightarrow{\text{rdiscw}} H_s \)

and \( \sigma_r \xrightarrow{\text{conc}} C \), where \( w \) is a fresh (new) index considering all edges of kind \( p \), \( \text{disc} \) and \( c \) ingoing and/or outgoing of the formula-nodes \( \alpha_m \), \( \beta_n \) and \( \rightarrow_q \); is a mimp-graph, see figure below.

\[ \Rightarrow \exists v, \land E, \lor I, \lor I \] Similar to other cases of construction (for an expanded version see [4]).

In the terminology about inference rules or R-nodes, when an R-node has more than one incoming edge, these are distinguished by calling them left, right, major or minor, or a combination of these terms and so also the F-node ‘premise’ associated with these edges. Thus, the major premise in R-node contains the connective that is eliminated; the other premise in R-node is called ‘minor’. Two premises that play a more or less equal role in the inference are called ‘left’ and ‘right’. For instance, an R-node \( \lor \) has a major premise, a left minor premise and a right minor premise; an R-node \( \land I \) has a left premise and a right premise.

The term \textit{R-node sequence} is representing a deduction, and if it is a smaller part of another R-node sequence (deduction), then it is called \textit{a subsequence} of the latter. A subsequence that derives a premise of the last R-node application in an R-node sequence is called a direct R-node subsequence. Instead of writing “the direct R-node subsequence that derives the minor premise of the last inference of an R-node sequence \( D \)”, we simply write “\textit{the minor subsequence of} \( D \)”.

\[ \odot \] By definition \( G_1 \odot G_2 \) equalizes or collapses the R-nodes of \( G_1 \) with the R-nodes of \( G_2 \) that have the same set of premises and conclusion keeping the inferential order of each node, and equalizes F-nodes of \( G_1 \) with the F-nodes of \( G_2 \) that have the same label, and equalizes edges with the same source, target and label into one.
Mimp-graphs need to conform a number of restrictions. To formulate the first one, acyclicity, we need the notion of inferential order on R-nodes that allows to pass through the nodes of the structure preventing nodes from infinitely reoccurring in a path.

**Definition 2.3** Let $G$ be a mimp-graph. An inferential order $<$ on nodes of $G$ is a partial ordering of the R-nodes of $G$ such that $n < n'$ iff $n$ and $n'$ are R-nodes and there is an F-node $f$ such that $n \xrightarrow{l_{\beta_1}} f \xrightarrow{l_{\beta_2}} n'$ and $l_{\beta_1}$ is $c$ and $l_{\beta_2}$ is $m$, or $l_{\beta_1}$ is $c$ and $l_{\beta_2}$ is $M$, or $l_{\beta_1}$ is $c$ and $l_{\beta_2}$ is $p$. Node $n$ is a top position node if $n$ is maximal w.r.t. $<$. 

**Definition 2.4** (1) For $n_i \in V$, a path in propositional mimp-graph is a sequence of vertices and edges of the form: $n_1 \xrightarrow{l_{\beta_1}} n_2 \xrightarrow{l_{\beta_2}} \ldots \xrightarrow{l_{\beta_{k-2}}} n_{k-1} \xrightarrow{l_{\beta_{k-1}}} n_k$, such that $n_1$ is a hypothesis formula node, $n_k$ is the conclusion formula node, $n_i$ alternating between a rule node and a formula node. The edges $l_{\beta_1}$ alternate between two types of edges: the first is $l_{\beta_1} \in \{r_m, l_m, m, M, rp, lp, ldisc\}$ and the second $l_{\beta_1} = c$. (2) A branch in propositional mimp-graph is an initial part of a path which stops at the conclusion F-node of the graph or at the first minor (or left) premise whose major (or right) premise is the conclusion of a rule node.

The Lemma 2.5 bellow enables us to prove that a given graph $G$ is a propositional mimp-graph. Among others it says that we have to check that each node of $G$ is of one of the possible types that generate the construction cases of Definition 2.2.

In order to avoid overloading of indexes, we will omit whenever possible, the indexing of edges of kind $r_m$, $l_m$, $lp$, $rp$, $ldisc$ and $rdisc$, keeping in mind that the coherence of indexing is established by the kind of rule-node to which they are linked.

**Lemma 2.5** $G$ is a propositional mimp-graph if and only if the following hold:

(i) There exists a well-founded (hence acyclic) inferential order $<$ on all rule nodes of the propositional mimp-graph.

(ii) Every node $N$ of $G$ is of one of the following ten types:

- **P** $N$ is labelled with one of the propositional letters: \{P, Q, R, ... \}. $N$ has no outgoing edges $l$ and $r$.
- **F** $N$ has one of the following labels: $\triangleright_i$, $\land_j$ or $\lor_k$, and has exactly two outgoing edges with label $l$ and $r$. $N$ has outgoing edges with labels $p$, $m$, $M$, $l_m$, $rm$, $lp$, $rp$; and ingoing edges with label $c$ and hyp.
- **E** $N$ has label $\rightarrow E_i$ and has exactly one outgoing edge $E_i \xrightarrow{l_{\beta_n}} \beta_n$, where $\beta_n$ is a node type $P$ or $K$. $N$ has exactly two ingoing edges $\alpha_m \xrightarrow{m} E_i$ and $q \xrightarrow{M} E_i$, where $\alpha_m$ is a node type $P$ or $K$. There are two outgoing edges from the node $q$: $q \xrightarrow{l} \alpha_m$ and $q \xrightarrow{r} \beta_n$.
- **I** $N$ has label $\rightarrow I_j$ (or $\rightarrow I_{jv}$, if discharges an hypothesis vacuously), has one outgoing edge $I_j \xrightarrow{l_{\beta_n}} \rightarrow$, and one (or zero for the case $\rightarrow I_{iv}$) outgoing edge $I_j \xrightarrow{disc_n} H_i$. $N$ has exactly one ingoing edge: $\beta_n \xrightarrow{p} I_j$, where $\beta_n$ is a node type $P$ or $K$. There are two outgoing edges from the node $\rightarrow I_j$: $\rightarrow \xrightarrow{l} \alpha_m$ and $\rightarrow \xrightarrow{r} \beta_n$ such that there is one (or zero for the case $\rightarrow I_{iv}$) ingoing edge to
the node $\alpha_m$: $H_k \xrightarrow{\text{hyp}} \alpha_m$.

I$^\land$ N has label $\land l$, one outgoing edge $\land l \xrightarrow{\text{c}} \land l$ and exactly two ingoing edges: $\alpha_m \xrightarrow{\text{lp}} \land l$ and $\beta_n \xrightarrow{\text{rp}} \land l$, where $\alpha_m$ and $\beta_n$ are nodes type P or F. There are two outgoing edges from the node $\land l$: $\land l \xrightarrow{r} \alpha_m$ and $\land l \xrightarrow{r} \beta_n$.

E$^\land$ N has label $\land E_i$, one outgoing edge $\land E_i \xrightarrow{\text{c}} \alpha_m$ where $\alpha_m$ (or $\beta_n$ in the case $\land E_i$) is a node type P or F and has exactly one outgoing edge: $\land E_i \xrightarrow{r} \land E_i$. There are two outgoing edges from the node $\land l$: $\land l \xrightarrow{r} \alpha_m$ and $\land l \xrightarrow{r} \beta_n$.

I$^\lor$ N has label $\lor I_i$, one outgoing edge $\lor I_i \xrightarrow{\text{c}} \lor I_i$ and has exactly one outgoing edge: $\alpha_m \xrightarrow{p} \lor I_i$ where $\alpha_m$ (or $\beta_n$ in the case $\lor I_i$) is a node type P or F. There are two outgoing edges from the node $\lor I_i$: $\lor I_i \xrightarrow{l} \alpha_m$ and $\lor I_i \xrightarrow{l} \beta_n$.

E$^\lor$ N has label $\lor E_i$, three outgoing edges $\lor E_i \xrightarrow{\text{c}} \sigma_r$, $\lor E_i \xrightarrow{\text{l} \text{disc}} H_u$ and $\lor E_i \xrightarrow{\text{r} \text{disc}} H_s$; and it has exactly three ingoing edges: $\lor I_i \xrightarrow{\text{M}} \lor E_i$, $\lor I_i \xrightarrow{\text{im}} \lor E_i$, $\lor I_i \xrightarrow{\text{rm}} \lor E_i$ where $\alpha_m$ (or $\beta_n$ in the case $\lor E_i$) is a node type P or F. There are two outgoing edges from the node $\lor I_i$: $\lor I_i \xrightarrow{l} \alpha_m$, $\lor I_i \xrightarrow{l} \beta_n$ and the hypothesis edges:

$H_u \xrightarrow{\text{hyp}} \alpha_m$ and $H_s \xrightarrow{\text{hyp}} \beta_n$.

H N has label $H_k$ and has exactly one outgoing edge hyp.

C N has label C and has exactly one outgoing edge conc.

Proof.

$\Rightarrow$: Argue by induction on the construction of propositional mimp-graph (Definition 2.2). For every construction case for propositional mimp-graphs we have to check the three properties stated in Lemma. Property (2) is immediate. For property (1), we know from the induction hypothesis that there is an inferential order $<$ on R-nodes of the propositional mimp-graph. In construction cases I, E, A, $\land El$, $\land Er$, $\lor I$, $\lor E$, we make the new R-node that is introduced highest in the $<$-ordering, which yields an inferential ordering on R-nodes. In the construction case $\land I$, when we have two inferential orderings, $<_1$ on $G_1$ and $<_2$ on $G_2$. Then $G_1 \oplus G_2$ can be given an inferential ordering by taking the union of $<_1$ and $<_2$ and in addition putting $n < m$ for every R-node $n,m$ such that $n \in G_1, m \in G_2$. In the construction case $\lor E$, when we have three inferential orderings, $<_1$ on $G_1$, $<_2$ on $G_2$ and $<_3$ on $G_3$. Then $(G_1 \cup G_2) \oplus G_3$ can be given an inferential ordering by taking the union of $<_1$, $<_2$ and $<_3$ and in addition putting $n < m < p$ for every R-node $n,m,p$ such that $n \in G_1, m \in G_2, p \in G_3$.

$\Leftarrow$: Argue by induction on the number of R-nodes of $G$. Let $<$ be the topological order that is assumed to exist. Let $n$ be the R-node that is maximal w.r.t. $<$. Then $n$ must be on the top position. When we remove node $n$, including its edges linked (if $n$ is of type $F'$) and the node type $C$ is linked to the premise of the R-node, we obtain a graph $G''$ that satisfies the properties listed in Lemma. By induction hypothesis we see that $G''$ is a propositional mimp-graph. Now we can add the node $n$ again, using one of the construction cases for propositional mimp-graphs: $\text{mimp}$ if $n$ is a L node, $F$ node, $\Rightarrow E$ node or $\rightarrow I$ node, $\text{F}$ if $n$ is a $\land I$ node, $\text{E}$ if $n$ is a $\land El$ node or $\land Er$ node, $\text{F}$ if $n$ is a $\lor I$ node or $\lor E$ node, $\text{E}$ if $n$ is a $\lor E$ node. \qed
3 Normalization for propositional mimp-graphs

3.1 Elimination of maximal formula

In this section, we describe the normalization process for propositional mimp-graphs. Eliminating a maximal formula is very similar to the procedure for mimp-graphs described in [5], where we considered only the case of implication, now we define the maximal formulas in conjunction, disjunction and implication. The notion of reordering is provided as well, because when the maximal formula is removed a reordering of nodes occurs.

Definition 3.1 A maximal formula $m$ in a propositional mimp-graph $G$ is a sub-graph of $G$ as follows:

- ∧I followed by ∧El. It is composed of (see Figure 5(a)):
  (i) the F-nodes: $\alpha_m$, $\beta_n$ and $\land_q$, where $\land_q$ has zero or more ingoing/outgoing edges\(^6\), e.g. $\land_q$ could be premise or conclusion of others R-nodes;
  (ii) the R-nodes: $\land_{Ii}$ and $\land_{El}$, where $\land_{Ii}$ has an inferential order lower than $\land_{El}$ and there are zero or more maximal formulas between them\(^7\). If these nodes occur in different branches, a branch must be insertable\(^8\) in the other branch or bifurcated by an R-node ∨E;
  (iii) the edges: $\land_q \xrightarrow{l} \alpha_m, \land_q \xrightarrow{r} \beta_n, \alpha_m \xrightarrow{l_p} \land_{Ii}, \beta_n \xrightarrow{r_p} \land_{Ii}, \land_{Ii} \xrightarrow{c} \land_q, \land_q \xrightarrow{p} \land_{El}$ and $\land_{El} \xrightarrow{c} \alpha_m$.

There is a symmetric case for ∨I followed by ∨Er.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{maximal_formulas}
\caption{Maximal Formulas}
\end{figure}

- ∨II followed by ∨E. It is composed of (see Figure 5(b)):
  (i) the F-nodes: $\alpha_m$, $\beta_n$, $\lor_q$ and $\sigma_x$, where $\lor_q$ has zero or more ingoing/outgoing edges\(^9\).

\(^6\) Represented in the figure by double-headed arrows
\(^7\) The maximal formulas are represented in the figure by nodes labelled with $I$ and $E$
\(^8\) A branch is insertable in other branch when it is bifurcated by a maximal formula: $\to I$ followed by $\to E$
\(^9\) Represented in the figure by double-headed arrows
(ii) the D-nodes: $H_r$ and $H_s$;
(iii) the R-nodes in ascending inferential order: $\lor I_i$, and there are zero or more maximal formulas in branches between them. If these nodes occur in different branches, a branch must be insertable in the other branch or bifurcated by an R-node $\lor E$;
(iv) the edges: $\lor q_l \rightarrow \alpha_m$, $\lor q_r \rightarrow \beta_n$, $\lor q_m \rightarrow \lor I_i$, $\lor q_i \rightarrow \lor E_l$, $\lor q_x \rightarrow \lor E_l$, $\lor q_{rdisc} \rightarrow \lor E_l$, and $\lor q_{disc} \rightarrow \lor E_l$, $\lor q_{disc} \rightarrow \lor E_l$.

There is a symmetric case for $\forall I_r$ followed by $\lor E$.

- $\forall I$ followed by $\forall E$. It is composed of (see Figure 5(c)):
  (i) the formula nodes: $\alpha_m$, $\beta_n$ and $\rightarrow q$, where $\rightarrow q$ has zero or more ingoing/outgoing edges;
  (ii) the D-node: $H_u$;
  (iii) the R-nodes in ascending inferential order: $\forall I_i$ and $\forall E_l$, and there are zero or more maximal formulas between them. If these nodes occur in different branches, a branch must be insertable in the other branch or bifurcated by an R-node $\lor E$;
  (iv) the edges: $\forall q_l \rightarrow \alpha_m$, $\forall q_r \rightarrow \beta_n$, $\forall q_m \rightarrow \forall I_i$, $\forall q_i \rightarrow \forall E_l$, $\forall q_x \rightarrow \forall E_l$, $\forall q_{rdisc} \rightarrow \forall E_l$, $\forall q_{disc} \rightarrow \forall E_l$.

Definition 3.2 A reordering of a given mimp-graph $G$ is obtained by supplying $G$ with the following (new) inferential order on the R-nodes of $G$.

- $o(t_m) = 0$ for an R-node $t_m$ starting with hypothesis.
- $o(t) = o(t') + 1$ if the conclusion formula of R-node $t'$ is premise, right premise or major premise of $t$.

Proposition 3.3 A graph obtained by a reordering according to Definition 3.2 is a mimp-graph.

Note that the actual situation is more complicated than those sketched in Figures 5(a), 5(b) and 5(c). There are five sub-cases for each maximal formula due to the presence of disjunction and other maximal formulas. For brevity we only show how subcases of the elimination of $\forall I_l$ followed by $\forall E$ are treated (for an expanded version see [4]).

Definition 3.4 Given a propositional mimp-graph $G$ with a maximal formula $m$, eliminating a maximal formula is the following transformation of a propositional mimp-graph:

Elimination of $\forall I_l$ followed by $\forall E$ There is a symmetric case for $\forall I_r$ followed by $\forall E$. The elimination of this maximal formula is the following operation on a mimp-graph:

(i) If there are no maximal formulas in branches between the R-nodes $\forall I_i$ and $\forall E_l$ then follow these steps:

---

10 The maximal formulas are represented in the figure by nodes labelled with $I$ and $E$
11 Represented in the figure by double-headed arrows
12 The maximal formulas are represented in the figure by nodes labeled with $I$ and $E$
(a) If \( \forall I_i \) and \( \forall E_l \) are not bifurcated by one \( \forall E \) then (see cases 1 and 2 in Figure 6).
 Remove the R-nodes \( \forall I_i \) and \( \forall E_l \), and their edges.
 If the F-node \( \forall q \) only has outgoing edges to sub-formulas then remove it (see case 2 in Figure 6).

(b) Else if \( \forall I_i \) represents two R-nodes then (see case 3 in Figure 7):
 Remove R-nodes \( \forall I_i \) and \( \forall E_l \), and their edges.
 Eliminate edges:
\[
\forall q_l \xrightarrow{l,m} \forall E_k, \quad \forall q_r \xrightarrow{r,m} \forall E_k, \quad \forall E_k \xrightarrow{c} \forall q.
\]
 If the F-node \( \forall q \) only has outgoing edges to sub-formulas then remove it (see case 4 in Figure 7).
 Add the edges:
\[
\sigma_l \xrightarrow{l,m} \forall E_k, \quad \sigma_r \xrightarrow{r,m} \forall E_k, \quad \forall E_k \xrightarrow{c} \sigma_x.
\]
 Incorporate the inference orders of sequence \( \Pi^m_x \) of the Figure 7 in the minor subsequence of \( \forall E_k \) (left and right).

(c) Else (see case 5 in Figure 8)
 Remove the R-node \( \forall I_i \), and its edges.
 Eliminate edges:
\[
\forall q_l \xrightarrow{l,m} \forall E_k, \quad \forall q_r \xrightarrow{r,m} \forall E_k, \quad \forall E_k \xrightarrow{c} \forall q.
\]
 Add the edges:
\[
\sigma_l \xrightarrow{l,m} \forall E_k, \quad \sigma_r \xrightarrow{r,m} \forall E_k, \quad \forall E_k \xrightarrow{c} \sigma_x.
\]
 Incorporate the inference order of node \( \forall E_l \) with its subsequences \( \Pi^m_x \) and \( \Pi^r_x \) as shown in Figure 8 in the right minor subsequence of \( \forall E_k \) and incorporate the R-node sequence \( \Pi^m_x \) in the left minor premise of \( \forall E_k \).

(d) Apply the operation defined in Definition 3.2 to the resulting graph. Note that Proposition 3.3 ensures that the result is a mimp-graph.

(ii) Otherwise eliminate the maximal formulas in branches between the R-nodes \( \forall I_i \) and \( \forall E_l \).

**Lemma 3.5** If \( G \) is a propositional mimp-graph with a maximal formula \( m \) and \( G' \) is obtained from \( G \) by eliminating \( m \), then \( G' \) is also a propositional mimp-graph. Moreover \( G \) and \( G' \) both have the same conclusion, i.e. the F-label being the source of conc.

**Proof.** We use Lemma 2.5. All nodes in \( G' \) are of the right form: \( P, K, E, I, E', I', E^\wedge, I^\wedge, H \) or \( C \). We verify that \( G' \) has one ingoing edge with label conc to the D-node
with label $C$ and that is acyclic and connected. Finally, an inferential order on $G'$ (as defined in Definition 3.2) between rule nodes must preserve the derivability and
the conclusions.

3.2 Normalization proof

This proof is guided by the normalization measure. That is, a given mimp-graph $G$ should be transformed into a non-redundant mimp-graph by applying reduction steps and at each reduction step the measure must be decreased. The normalization measure will be the number of maximal formulas in the mimp-graph.

**Theorem 3.6 (Normalization)** Every propositional mimp-graph $G$ can be reduced to a normal propositional mimp-graph $G'$ having the same hypotheses and conclusion as $G$. Moreover, for any standard tree-like natural deduction $\Pi$, if $G := G_\Pi$ (the F-minimal mimp-like representation of $\Pi$), then the size of $G'$ does not exceed the size of $G$, and hence also $\Pi$.

**Remark 3.7** The second assertion sharply contrasts to the well-known exponential speed-up of standard normalization. Note that the latter is a consequence of the tree-like structure of standard deductions having different occurrences of equal hypotheses formulas, whereas all formulas occurring in F-minimal mimp-like representations are pairwise distinct.

**Proof.** This characteristic of preservation of the premises and conclusions of the derivation is proved naturally. Through inspection of each elimination of maximal formulas it is observed that the reduction step (see Definition 3.4) of the propositional mimp-graph does not change the set of premises and conclusions (indicated by the D-nodes $H$ and $C$) of the derivation that is being reduced.

In addition, the demonstration of this theorem has two primary requirements. First, we guarantee that through the elimination of maximal formulas in the propositional mimp-graph, cannot generate more maximal formulas. The second requirement is to guarantee that during the normalization process, the normalization measure adopted is always reduced.

The first requirement is easily verifiable through an inspection of each case in the elimination of maximal formulas. Thus, it is observed that no case produces more maximal formulas. The second requirement is established through the normalization procedure (see Section 3.2.1) and demonstrated through an analysis of existing cases in the elimination of maximal formulas in mimp-graphs. To support this statement, it is used the notion of normalization measure, we adopt as measure of complexity (induction parameter) the number of maximal formulas $Nmax(G)$. Besides, as already mentioned, working with F-minimal mimp-graph representations we can use as optional inductive parameter the ordinary size of mimp-graphs.

3.2.1 Normalization Process

We know that a specific propositional mimp-graph $G$ can have one or more maximal formulas represented by $M_1, \ldots, M_n$. Thus, the normalization procedure is:

(i) Choose a maximal formula represented by $M_k$.

(ii) Identify the respective number of maximal formulas $Nmax(G)$.

(iii) Eliminate $M_k$ as defined in Definition 3.4, creating a new graph $G$.
(iv) In this application one of the following six cases may occur:

a) The maximal formula is removed (case 1 in all eliminations of maximal formulas).

b) The maximal formula is removed but the formula node is maintained, and,
   \( N_{max}(G) \) is decreased (case 2 in all eliminations of maximal formulas);

c) Two maximal formula are removed (case 3 in all eliminations of maximal formulas).

d) Two maximal formula are removed but the formula node is maintained, hence
   \( N_{max}(G) \) is decreased (case 4 in all eliminations of maximal formulas).

e) The maximal formula is removed, the formula node is maintained and R-node sequence reordered, hence
   \( N_{max}(G) \) is decreased (case 5 in all eliminations of maximal formulas).

f) All maximal formulas are removed.

(v) Repeat this process until the normalization measure \( N_{max}(G) \) is reduced to
   0 and \( G \) becomes a normal propositional mimp-graph.

Since the process of the eliminating a maximal formula on propositional mimp-graphs always ends in the elimination of at least one maximal formula, and with the decrease in the number of vertices of the graph, we can say that this normalization theorem is directly a strong normalization theorem.

4 Conclusions

The results presented for mimp-graph in [5] are extended for propositional mimp-graph. Thus, propositional mimp-graph was introduced through definitions and examples preserving the ability to represent proofs in Natural Deduction. The minimal formula representation is a key feature of the mimp-graph structure, because as we saw earlier, it is easy to determine maximal formulas and upper bounds in the length of reduction sequences to leading to normal proofs. A normalization theorem was proved by counting the number of maximal formulas in the original derivation. The strong normalization property is a direct consequence of such normalization, since any reduction decreases the corresponding measures of derivation complexity. This is a preliminary step into investigating how a theorem prover based on graphs is more efficient than usual theorem provers.

References


Normalization of N-Graphs via Sub-N-Graphs

Laís Andrade
Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil

Ruan Carvalho
Dept. de Estatística e Informática
Universidade Federal Rural de Pernambuco
Recife, Brazil

Anjolina de Oliveira
Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil

Ruy J. G. B. de Queiroz
Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil

Abstract

Alves presented in his PhD thesis a normalization procedure for N-Graphs, a multiple conclusion natural deduction for propositional classical logic proposed by de Oliveira in 2001, with proofs as directed graphs. Here we develop a new normalization for N-Graphs inspired by A. Carbones work in 1999, where she proposed a combinatorial model to study the evolution of proofs during the procedure of cut elimination.

Keywords: N-Graphs, Normalization, Directed graphs, Duplication

1 Introduction

Whenever one is concerned with the study of proofs from a geometric perspective one can hardly overestimate the pioneering work of Statman in his doctoral thesis Structural Complexity of Proofs [19]. Drawing on Statman’s legacy, for the
last three decades at least two research programmes have approached the study of structural properties of formal proofs from a geometric perspective: (i) the notion of proof-net, given by Girard in [12] in the context of linear logic; and (ii) the notion of logical flow graph given by Buss in [6] and used as a tool for studying the exponential blow up of proof sizes caused by the cut-elimination process, in this case giving rise to a programme (1996–2000) proposed by Carbone in collaboration with Semmes [7]. Statman’s geometric perspective has given an important legacy, namely the idea of extracting structural properties of proofs in natural deduction (ND) using appropriate geometric tools and intuitions. The lack of symmetry in ND presents a challenge for such a kind of study. Of course, the obvious alternative is to look at multiple-conclusion calculi. One can find in the literature different approaches involving such calculi, such as, for example, Kneale’s tables of development [13] (studied in depth by Shoesmith & Smiley [18]) and Ungar’s multiple-conclusion ND [20]. But then a great challenge remained: normal forms and the normalization procedure.

The system of N-Graphs, a multiple conclusion ND developed in the early 2000’s by de Oliveira [14] out of a combination of the techniques developed in the two aforementioned research programmes, has revealed itself as a rather appropriate framework in which to formulate and explore techniques for normalizing ND proofs in the form of directed graphs. N-Graphs were motivated by the idea of proofs as geometric objects and aimed towards the study of the geometry of Natural Deduction systems. Following that line of research, we propose a normalization procedure defined as a set of combinatorial operations on graphs that can offer a framework for future combinatorial studies on the proof growth during normalization. The procedure we present in this paper also works as an extension of the normalization defined by Prawitz, i.e. it enjoys the separation and subformula properties.

In her analysis of the blow-up of proof-size after cut elimination in sequent calculus proofs, Carbone defined an operation called duplication and worked with the logical flow graphs extracted from sequent calculus proofs in order to propose a purely combinatorial analysis of cut elimination [8]. Our procedure uses Alves’ original beta and permutative reductions [2], changing reductions with the link with meta-edge. A new set of switchable reductions is presented, combined with an adaptation of the duplication operation for sub-N-Graphs, to handle switchable links. As a result, this new normalization has a stronger parallel with both Prawitz’s normalization and Gentzen’s cut-elimination, offering a good start point for studies on correspondences between those two procedures, like the ones presented by Zucker [21] and Ungar [20]. This also offers a base to extend Carbone’s results on proof growth during cut-elimination in sequents to a ND system.

2 N-Graphs

Proposed by de Oliveira [14,15], N-Graphs is a symmetric natural deduction (ND) calculus with the presence of structural rules, similar to the sequent calculus. It is a multiple conclusion proof system for classical logic where proofs are built in the form of directed graphs (“digraphs”). Several studies have been developed on N-Graphs since its first publication in 2001 [14], like Alves’ development on the
geometric perspective and cycle treatment towards the normalization of the system
[3] and Cruz’s definition of intuitionistic N-Graphs [9]. A normalization algorithm
was presented for classical N-Graphs [1], along with the subformula and separation
properties [2]. Also, a linear time proof checking algorithm was proposed [4], and
more recently a new sequentialization proof was presented [10], using an adaptation
of the concept of subnets from proof-nets to create the sub-N-Graphs and perform
cuts in classical logic proofs with the presence of defocussing switchable links and
no axiom links.

2.1 Proof-Graphs

The system is defined somewhat like proof-nets. There is the concept of proof-
graphs, from which all graphs are constructed with the valid links where each node
is the premise and conclusion of at most one link, and the concept of N-Graphs,
which are the correct proof-graphs, i.e. the proof-graphs that represent valid proofs.
These constructions are analogous to the definition of proof-structure and proof-net,
respectively[12].

The links represent atomic steps in a derivation. Focussing links are the ones
with two premises and one conclusion, as illustrated by Fig. 1 (∧−I, ⊥−link, →−E,
⊤−focussing weak and contraction). The defocussing links are the ones with one
premise and two conclusions, as shown in Fig. 1 (∨−E, ⊤−link, →−I, ⊥−defocussing weak
and expansion). All other links are called simple links and have only one premise
and one conclusion (Fig. 1).

There are two kinds of edges (“solid” and “meta”) and the second one are labeled
with an “m” ((u,v)m). The solid indegree (outdegree) of a vertex v is the number
of solid edges oriented towards (away from) it. The meta indegree and outdegree
are defined analogously. The set of vertices with indegree (outdegree) equal to zero
is the set of premises (conclusions) of the proof-graph G, and is represented by
PREMIS(G) (CONC(G)). The set of vertices with solid indegree equal to zero and
meta indegree equal to one is the set of canceled hypothesis of G (HYPOT(G)).

A logical link represents a derivation in ND, according to its name (⊤−link acts
as the law of the excluded middle). A structural link expresses the application of
a structural rule as it is done in sequent calculus: it enables weakening a proof
(⊤−focussing weak, ⊥−defocussing weak, ⊤−simple weak and ⊥−simple weak), duplicating
Fig. 2. Meta edge: an invalid application on the left for \( \vdash A \lor B \) and a sound one on the right for \( \vdash (A \lor B) \rightarrow A, B \).

premises (expansion link) and grouping conclusions in equivalence classes (contraction link). There is no link to emulate the interchange rule because in a proof-graph the order of the premises is not important for the application of derivation rules.

The axioms are represented by proof-graphs with one vertex and no edges. Then, a single node labeled by \( A \) is already a valid derivation: it represents an axiom in sequent calculus (\( A \vdash A \)).

2.1.1 Meta-edge and the scope of the hypothesis
Both Ungar and Gentzen’s systems are formulated in such a way that when the \( \rightarrow \) connective is introduced it may eliminate an arbitrary number of premises (including zero). In N-Graphs this introduction is made in a more controlled way, which also complicates the task of identifying inadequate proof-graphs. For example, the first proof in Fig. 2 is not correct, but the second one is.

2.2 Soundness criteria
Similar to Danos-Regnier’s criterion [11], we define the following subgraphs associated to a proof-graph.

Definition 2.1 [Switching] Given a proof-graph \( G \), a switching graph \( S(G) \) associated with \( G \) is a spanning subgraph\(^5\) of \( G \) in which the following edges are removed: one of the two edges of every expansion link and one of the two edges of every contraction link.

Definition 2.2 [Meta-switching, virtual edge] Given a proof-graph \( G \), a meta-switching graph \( S(G) \) associated with \( G \) is a switching of \( G \) in which every link with meta-edge \( \{ (u, w), (u, v)^m \} \) is replaced by one of the following edges: the one from \( u \) to \( w \) or an edge from \( v \) to \( w \), which is defined as virtual edge.

Definition 2.3 [N-Graph derivation] A proof-graph \( G \) is a N-Graph derivation (or N-Graph for short) iff every meta-switching graph associated with \( G \) is acyclic and connected.

Contraction and expansion links are fundamentals in the soundness criteria: the formulas connected by them in a proof-graph must be already connected some other way in order to the proof to be sound. The \( \rightarrow \)-I also plays an important role: the premise of the link (\( B \)) and the canceled hypothesis (\( A \)) need to be already connected some other way in the proof so it can be sound. Thus the meta-switching must choose to connect \( A \rightarrow B \) to \( A \) or \( B \), and the resulting subgraph must be connected.

\(^5\) A spanning subgraph is a subgraph \( G_1 \) of \( G \) containing all the vertices of \( G \).
and acyclic no matter the choice. In the first proof-graph of Fig. 2 the conclusion of \( \rightarrow \neg I \) is \( A \rightarrow (A \lor B) \), so this formula already carries a dependency on \( A \) and the meta-edge removes the node from the set of premises of the proof. However, there is another occurrence of \( A \) that comes from the same initial node, which is used by the \( \rightarrow \neg E \) link to obtain a ‘proof’ of \( \vdash A \lor B \).

The soundness and completeness of the system were proved through a mapping between N-Graphs and \( LK \) (sequent calculus for classical logic) [14,15,10].

3 Normalization of N-Graphs

Alves presented in his PhD thesis [2] a normalization procedure for N-Graphs. Inspired in the work of Statman [19], Blute et. al. [5] and others that created graphical and topological frameworks to study the normalization of natural deduction, he devised a set of operations divided in two stages: the first that handles general proof-graphs, i.e. trees and some specific cycles, and the second stage that was constructed to give a thorough treatment of cycle structures. In the end, the proposed procedure has four sets of transformations that eliminate maximum formulas/segments and an algorithm (named \( 3CA \)) designed to determine whether a cycle has a detour or not.

Here we present a different normalization procedure with some of the reductions defined by Alves (\( \beta \) and permutative weakening reductions) slightly modified and new approaches for the permutative switchable reductions. The topological framework devised to check and transform cycle structures to remove detours was abandoned, but the separation and subformula properties can still be proved for normalized N-Graphs. This new and simpler normalization also uses an operation defined by Carbone [8] called duplication, leading to an extension of her work that can generate a combinatorial model for the study of the proof size during the normalization of N-Graphs.

We call main formula the formula in a reduction between the two links, and peripheral formulas all other formulas from both links. The reduction images are illustrative, and the graphs represented by \( G_i, \ i \in [1,4] \) may be connected (or, in the switchable reductions, some of them must be connected). The switchable links (contractions and expansions) are illustrated with dotted undirected edges to keep the images cleaner and also highlight their occurrences.

3.1 \( \beta \) reductions

Definition 3.1 [Maximum formula] A formula occurrence \( A \) is a maximum formula in an N-Graph \( G \) if it is the conclusion of an I-flavour link and the premise of an E-flavour link.

The fact that N-Graphs is a multiple conclusion calculus affects the nature of the reductions in a structural way that does not happen with the reductions introduced by Prawitz [16] for Natural Deduction. Once the proofs are represented by graphs, not only trees, the pruning that happens in the proof during the elimination of a maximum formula of the type \( A \lor B \), for example, cannot be replicated in a multiple
conclusion system where the geometrical structure of the proof (connectivity) is fundamental for its correctness.

The $\beta$ reductions that eliminate maximum formulas have then a “conservative” aspect, as can be seeing in Fig. 3. Additionally, the presence of $\top$ and $\bot$ links raises a new kind of maximum formula where a non-atomic formula is a conclusion (premise) of a $\bot$ ($\top$) link and is followed (preceded) by an elimination (introduction) rule. These aspects of maximum formulas have been presented by Ungar in [20]. We then call all introduction links and $\bot$ link as $I$-flavour links, and elimination links and $top-link$ as $E$-flavour links [2]. One last kind of $\beta$ reduction removes a $\bot$-link followed by a $\top$-link, removing from the proof what Alves described as a “hole” that compromises the subformula property.

3.2 Permutative weakening reductions

In Prawitz’s Natural Deduction a concept of “maximum segment” is defined to address the maximum formulas that might be hidden in the proof by the propagation of a formula in the tree. It happened exclusively in the $\lor$ and $\exists$ elimination rules that use a side formula $C$ to perform the elimination of the connective. Multiple conclusion natural deduction systems usually remove the need of such formula creating a system where introduction and elimination rules are symmetric, somewhat like in sequent calculus. N-Graphs is no different, but these segments may still arise in a proof by the application of structural rules to perform weakening.
Definition 3.2 [Segment, Maximum segment, Structural segment] A segment from the vertex \( u \) to the vertex \( v \) in an N-Graph \( G \) is a sequence of directed edges \((u_0, v_0), (u_1, v_1), \ldots, (u_n, v_n)\), where \( u_0 = u, v_n = v, v_i = u_{i+1} \). A segment from the occurrence \( u \) of the formula \( A \) to the occurrence \( v \) of the same formula in an N-Graph \( G \) is a maximum segment if \( u \) is the conclusion of an I-flavour link, \( v \) is the premise of an E-flavour link and every other edge in the segment is part of a structural link. The same segment is a structural segment if \( u \) is the conclusion of a weakening link and \( v \) is the premise of an I-flavour/expansion link, or if \( u \) is the conclusion of an E-flavour/contraction link and \( v \) is the conclusion of a weakening link.

Here enters the permutative reductions on weakening formulas. The focus of these transformations is to move down I-flavour links and move up E-flavour links (up and down here following the directions of the edges in the directed graph). The maximum formulas that might be hidden in the proof will become explicit and then they can be removed by \( \beta \) reductions. In Fig. 4 we can see the reductions for introduction rules and focussing weakening links. The other reductions can be seen in Appendix A.

![Fig. 4. Permutative reductions: introduction links followed by focussing weakenings.](image)

3.3 Permutative switchable reductions

The switchable reductions are the ones responsible for the exponential blow up of the proof during the normalization procedure. Expansions and contractions can be part of a maximum segment, as they are also structural links, and thus need to be permutated with I/E-flavour rules to show possible hidden maximum formulas. Due to the switchable aspect of those links, the permutation needs to be taken with care in order to avoid the transformation of a valid proof-graph into an invalid one.

Before we present the reductions, we need to define the duplication operation for proof-graphs [8]:

Definition 3.3 [Duplication] The duplication \( D \) is a binary operation applied to a proof-graph \( G \) and a subgraph \( G' \) of \( G \) with the property that:

(i) if a vertex of \( G' \) is a focussing point in \( G \), then either its immediate predecessor vertices both lie in \( G' \) or none of them does;

(ii) if a vertex of \( G' \) is a defocussing point in \( G \), then either its immediate successor vertices both lie in \( G' \) or none of them does;

(iii) at least one premise or conclusion in \( G' \) is both premise and conclusion of switchable links or \( \rightarrow I \).
The duplication of $G'$ in $G$ is a graph $D(G, G')$ defined as $G$ except on $G'$, which will be replaced by two copies of it and the following extra vertices:

(i) let $u$ be a premise in $G'$ (i.e., no edge towards it in $G'$) and $u_1, u_2$ its copies in $D(G, G')$. Then a new vertex $u'$ will be created and linked to $u_1$ and $u_2$ with an expansion. If there are edges in $G$ towards $u$, then link those edges to $u'$ in $D(G, G')$.

(ii) let $v$ be a conclusion in $G'$ (i.e., no edge from it in $G'$) and $v_1, v_2$ its copies in $D(G, G')$. Then a new vertex $v'$ will be created and linked to $v_1$ and $v_2$ with a contraction. If there are edges in $G$ from $v$, then link those edges to $v'$ in $D(G, G')$.

The exception to the procedure above is the link from item 3. In this case, we will collapse the vertices $u_1$ and $u_2$ in $G - G'$ with the copies $v_1$ and $v_2$ of $v$ in $G'$.

This operation is a little different from the one Carbone defined for optical flow graphs. The difference is on how the two copies of $G'$, namely $G'_1$ and $G'_2$, will be attached to the original vertices in $G$. In Fig. 5 we have an illustration of the duplication operation for proof-graphs.

![Fig. 5. Duplication of a proof-graph: the white and 1,2 vertices are boundary points, the dotted edges linked to 1 and 2 are the dissolved edges and 1 and 2 on the left are the collapsed vertices.](image)

**Definition 3.4** [boundary [8], dissolved edge, collapsed vertex] Let $G$ be a graph and $G'$ be a subgraph of it. We say that a point in $G$ is a boundary point if it does not belong to $G'$ but is linked to points of $G'$. The dissolved edge $(u, v) \in G$ is an edge removed from $D(G, G')$ between $u$ (in the boundary) and $v$ (in $G'$), where $u$ was collapsed with the collapsed vertex $v_i$ from $G'_i$. In other words, the collapsed vertices are the ones in the boundary that belongs to $G'_i$.

**Theorem 3.5 (Duplication of sub-N-Graphs)** Let $G$ be an N-Graph and $G'$ a sub-N-Graph of $G$. Then $D(G, G')$ is an N-Graph.

**Proof.** Proof by contradiction. Lets assume $D(G, G')$ is not an N-Graph. Then we have two cases:

(i) There is a disconnected meta-switching $S(D(G, G'))$:

As we know $G$ and $G'$ are N-Graphs, $S(D(G, G'))$ is not disconnected in any of the duplicated components. Also, there is always a path in $S(G - G')$ between the collapsed vertices, as those vertices are connected in $G$ by a switchable link. Let $u$ and $v$ be two disconnected vertices in $S(D(G, G'))$, and $\pi$ the
path in $S(G)$ from $u$ to $v$. Then, $\pi \cap G' \neq \emptyset$, otherwise $\pi$ would also be present in $S(D(G, G'))$. As those vertices are disconnected in $S(D(G, G'))$, $\pi'$ (the conversion of $\pi$ following the duplication definition) must connect $u$ to a vertex $w_1$ in $G'_1$ and $v$ to the copy $w_2$ in $G'_2$. But, as each copy $G'_i$ is connected in $S(G'_i)$, they are also connected to the respective collapsed vertex, and thus connected through $S(G - G')$.

(ii) There is a meta-switching $S(D(G, G'))$ with a cycle $c$.

As we know $G$ and $G'$ are N-Graphs, $S(D(G, G'))$ have no cycle in any of the duplicated component $G'$ nor in $G - G'$. Then the connection between $G - G'$ and the copies must create $c$. We have two cases:

(a) $c$ is in $G - G'$ and $G'_1$, a copy of $G'$:

We can construct the corresponding cycle $c'$ in $S(G)$ by doing the reverse of the duplication procedure: removing the added vertices and linking the boundary nodes to the original ones in $G'$; separating the collapsed vertex $v_i$ in $c$ into the original vertices $u, v$ and adding back the dissolved edge $(u, v)$. As $G$ is an N-Graph this cycle cannot exist, thus we arrived at a contradiction.

(b) $c$ is in $G - G'$ and both $G'_1$ and $G'_2$ copies of $G'$:

As $c$ goes through $G'_1$ and $G'_2$, and the only connection between those two copies is through $G' - G'$, there must be a path $\pi_1 \in c \cap G - G'$ from $u$ to $v$, both in the boundary. As $G'$ is an N-Graph, there must be a path $\pi_2$ in $G'$ from $u'$ to $v'$, where $(u, u') \in G$ and $(v, v') \in G$ (possible empty if $u = v$, i.e. $\pi_1 = c$). Then, we can create a cycle $c'$ in $S(G)$ by linking those two paths with the edges $(u, u')$, $(v, v')$, once again arriving at a contradiction.

In Fig. 6 we can see the reduction for the permutation of introduction rules with the contraction, and also the special case where a contraction is followed by an expansion. All other permutative switchable reductions can be found in Appendix A.

![Fig. 6. Switchable reductions: contractions followed by eliminations (first and second columns) and contractions followed by expansion (third column).](image)
an expansion. In this case we need to duplicate the north (south) empire of the main formula to eliminate an expansion (contraction). These operations are represented in the images by the *.

### 3.4 Normalization

**Definition 3.6** [Cut formula] A formula $A$ in an N-Graph $G$ is a cut formula if it is the main formula of a reduction.

**Definition 3.7** [Normal N-Graph] An N-Graph $G$ is normal iff there is no maximum formula and no maximum or structural segment. In other words, there is no cut formula in $G$.

**Theorem 3.8** (N-Graph normal form) A segment from a premise or discharged hypothesis $A$ to a conclusion $B$ of a normal N-Graph $G$ can be divided in three unique parts:

(i) elimination part, where each edge is part of an elimination link or an expansion.
(ii) weak part, where each edge is part of a weakening link.
(iii) introduction part, where each edge is part of an introduction link or a contraction.

The weak part is also divided into tree parts, the first with E-flavour links, the second with focussing and defocussing links, and the third one with I-flavour links.

**Proof.** The proof is a thorough examination of all reductions presented in this paper, considering all possible inversions:

(i) expansion after structural link: Figs. A.5 and A.7.
(iii) expansion after contraction: Fig. 6.
(iv) E-flavour after contraction: Fig. 6.
(v) expansion after I-flavour: Figs. A.6 and A.7.
(vi) I-flavour after E-flavour: Fig. 3.
(vii) structural after contraction: Figs. A.5 and A.7.
(viii) structural after I-flavour: Figs. 4, A.1.

It is simple to check every reduction reduces an N-Graph $G$ to another N-Graph $G'$. The Theorem 3.5 is enough to validate the reductions with duplication of empires. All other reductions can be validated by a careful examination of every switching $S(G')$, using the fact that the original $S(G)$ is acyclic and connected.

**Corollary 3.9** (Subformula property) Each formula in a normal N-Graph $G$ is a subformula of a formula in the set of premises or in the set of conclusion of $G$.

The subformula property is an extension of the normal form, once we notice no formula is introduced and then eliminated in any segment in the N-Graph.

With the reductions presented earlier we can prove the normalization theorem, which is stronger than the normal form theorem but weaker than the strong normalization theorem, following the definitions presented by Prawitz [16].
Theorem 3.10 (Normalization) Every N-Graph derivation reduces to normal form.

Proof. [sketch] When an N-Graph $G$ is reduced to $G'$, a maximum formula is removed or the length of a maximum/structural segment decreases. Even if new cut formulas are added by some reduction, we can choose an order to apply the reductions so the degree of the formula in the vertex is always smaller. This can be easily done for the simple reductions by choosing the cut formula with greatest degree, as the peripheral formulas have lower degree than the main formula of the reduction.

The reductions with duplication are a little more complicated, once it can duplicate a cut formula with the same degree as the main formula. In this case, it is possible to create an infinite reduction sequence. In order to avoid this we need to select not only the reduction with greatest degree, but also with a north (or south) empire to be duplicated with only cut formulas of lower degree. To prove it is always possible to find such cut formula we only need to show it is impossible to have a cycle as illustrated in Fig. 7.

![Fig. 7. Cycle of cut formulas, but north and south empires links can be arbitrary.](image)

The impossibility of such cycle comes from the application of nesting lemmas defined by Carvalho [10]: suppose such cycle exists, we arrive at the conclusion $eA^* \subsetneq eA^*$, $* \in \{\lor, \land\}$ for all formulas $A$ in the cycle. \qed

4 Conclusion

We have presented a new normalization procedure for N-Graphs, based only on cut formula reductions to remove maximum formulas and maximum/structural segments. The normal form we arrive at after a sequence of reductions is very similar to the one Prawitz defined in [16], with an analytical and synthetical part represented by the elimination and introduction parts, but stronger as it also fixates weakening links in the weak part of the normal form.

This normal form goes a step closer to the definition of an equivalence relation between N-Graph proofs. Prawitz suggested that an identity relation between derivations could be characterized in terms of reductions [17], and it works well for a proof system without weakening rules like ND. Weakening is essentially a way to combine derivations together and dispensing with some premises or conclusions, so it is permutative in nature with other deduction rules and can make it difficult to define such equivalence relation. The normal form we arrived at with our normalization fixates a position for the weakening rules, and thus facilitates the definition of an identity relation.
Proofs with cuts are shorter than cut-free ones. In our procedure this growth happens when a cut is hidden by a vertex that is both the conclusion and premise of switchable links (contraction, expansion and $\rightarrow$ $\leftarrow$). We need to duplicate a subproof in order to make the maximum formula explicit. This shows how the structural links in N-Graphs can bring some properties from cut-elimination into a natural deduction system. As we use a modified version of the combinatorial operation developed by Carbone [8] to study the permutation of cut rules with contractions in sequent calculus, its behaviour in N-Graphs is similar in combinatorial terms and also reflects the substitution of hypothesis from Prawitz [17]. This work is then taking a first step into a combinatorial study of the blow up of proof size in a natural deduction proof system.

References


A Other normalization reductions

Fig. A.1. Permutative reductions: introduction links followed by defocussing weakenings.

Fig. A.2. Permutative reductions: focussing weakening links followed by eliminations.
Fig. A.3. Permutative reductions: focussing/defocussing weakening links with $\top$-link and $\perp$-link.

Fig. A.4. Permutative reductions: defocussing weakening links followed by eliminations.
Fig. A.5. Permutative reductions: focussing/defocussing weakening links with expansions and contractions.

Fig. A.6. Switchable reductions: introduction links followed by expansions.

Fig. A.7. Switchable reductions: contractions and expansions with $\top$-link and $\bot$-link.
On Graphs for Intuitionistic Modal Logics

Paulo A. S. Veloso1,2
Progr. Eng. de Sistemas e Computação, COPPE
Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro, RJ, Brazil

Sheila R. M. Veloso3
Dept. Eng. de Sistemas e Computação, Fac. Engenharia
Universidade do Estado do do Rio de Janeiro (UERJ)
Rio de Janeiro, RJ, Brazil

Abstract
We present a graph approach to intuitionistic modal logics, which provides uniform formalisms for expressing, analysing and comparing Kripke-like semantics. This approach uses the flexibility of graph calculi to express directly and intuitively possible-world semantics for intuitionistic modal logics. We illustrate the benefits of these ideas by applying them to some familiar cases of intuitionistic multi-modal semantics.

Keywords: Intuitionistic modal logics, semantics, graph formulations, calculi, refutation, special relations.

1 Introduction
We present a graph approach to intuitionistic modal logics, which provides a flexible and uniform tool for expressing, analysing and comparing possible-world semantics.

This graph approach can be regarded as a version of diagrammatic reasoning, where we can express formulas by diagrams, which can be manipulated to unveil properties (like consequence and satisfiability). Graph representations and transformations, having precise syntax and semantics, give proof methods. An interesting feature of this approach is its 2-dimensional notation providing pictorial representations that support manipulations [3]. These ideas have been adapted to refutational reasoning [9] and applied to multi-modal classical logics [10].

Intuitionistic modal logic is an interesting and intriguing subject [5,7]. There seems to be little consensus on the appropriate approach to its semantics, witness the diversity of Kripke-like semantics proposed (see [4,8] and references therein).
Modal logics and graphs are closely connected. Kripke semantics can be presented via directed labelled graphs for the accessibility relation of each modality \[1\]. It is natural to represent that \(a\) is related to \(b\) via relation \(r\) by an arrow \(a \xrightarrow{r} b\).

We provide graph calculi having diagrams as terms and whose rules transform diagrams, capturing graphically the semantics of the modal operators and accessibility relations. These calculi provide uniform and flexible formalisms where one can explore Kripke-like semantics for intuitionistic modal logics: satisfaction conditions, valid formulas, etc. We illustrate these ideas by 2 case studies: the logics of \([8,4]\).

We will consider a modal language \(\mathcal{M}\) with set \(\Phi\) of formulas, given by sets \(\mathcal{P}\), of propositional letters, and \(\mathcal{R}\), of 2-ary relation symbols. The formulas of \(\mathcal{M}\) are generated by the grammar \(\varphi ::= \bot \mid p \mid \neg \varphi \mid \varphi' \land \varphi'' \mid \varphi' \lor \varphi'' \mid \varphi' \rightarrow \varphi'' \mid \langle r \rangle \varphi \mid \{r\} \varphi\).

2 Graphs and Modalities: Basic Ideas

We now introduce informally some basic ideas about graphs and modalities.\(^4\)

A graph amounts to a finite set of (alternative) slices. A slice \(S\) consists of an underlying draft \(\hat{S}\) together with a distinguished node (marked, e.g. \(\hat{w}\)). A draft amounts to a finite sketch. A sketch amounts to sets of nodes and arcs. Slices and graphs represent sets of states, whereas sketches describe restrictions on states.

Arrows may be binary or unary. A binary arrow stands for accessibility between states; we represent that node \(v\) is accessible from node \(u\) by the relation of \(r\) by a solid arrow labelled \(r\) from \(u\) to \(v\): \(u \xrightarrow{r} v\) (abbreviated \(u \xrightarrow{r} v\)). A unary arrow is meant to capture the fact that a formula holds at a state; we represent that formula \(\varphi\) holds at node \(w\) by a dashed line from \(w\) to \(\varphi\): \(w \xrightarrow{\neg \varphi}\) (abbreviated \(w |\neg \varphi\)).

Expressions will encompass slices, graphs and their complements (noted by an overbar). As such, an expression represents a set of states; so we can use unary arcs of the form \(w \xrightarrow{\neg \varphi} E\), where \(E\) is an expression.

We now introduce some concepts to be used and illustrated in Example 2.1.

A (sketch) morphism is a node mapping that preserves arcs. A (slice) homomorphism is a morphism of their underlying drafts that preserves distinguished nodes. A sketch may have conflicts that prevent its satisfaction. We consider two kinds of conflicts. One concerns contradictory 1-ary arcs: if sketch \(\Sigma\) has the pattern \(E \xrightarrow{\neg \neg w \neg \varphi} E\), then expression \(E\) is a witness of a conflict at node \(w\). If \(\Sigma\) has 1-ary arc \(w \xrightarrow{\neg \varphi} Q\), slice \(Q\) will be a witness of a conflict at node \(w\) if there is a morphism from \(Q\) to \(\Sigma\) mapping the distinguished node of \(Q\) to \(w\).

To reason about modal formulas, we convert them to expressions (with the same meaning) and reason graphically about these. The next example illustrates this approach. We reduce consequence to unsatisfiability: "every state satisfying \(\psi_1, \ldots, \psi_n\) also satisfies \(\theta]\) (noted \(\psi_1, \ldots, \psi_n \models \theta\) is equivalent to "there is no state satisfying \(\psi_1, \ldots, \psi_n\) and failing to satisfy \(\theta\) (noted \(\psi_1, \ldots, \psi_n, \overline{\theta} \models \bot\). Note that \(\neg\) is complementation rather than intuitionistic negation.

**Example 2.1** (Consequence via slice conversions) We reduce \(\langle r \rangle (\psi \land \theta) \models \langle r \rangle \psi\) to \(\{\langle r \rangle (\psi \land \theta), \langle r \rangle \psi\} \models \bot\). We first indicate the graph-calculus steps.

---

\(^4\) These and other ideas will be formulated more precisely later on.
(i) We convert \( \langle r \rangle \psi \) to expression E:

\[
\langle r \rangle \psi \approx \hat{w} r \to \to z \searrow \swarrow \psi
\]

(ii) We also convert \( \langle r \rangle (\psi \land \theta) \) to slice S as follows:

\[
\langle r \rangle (\psi \land \theta) \approx \hat{u} r \to \to v \searrow \swarrow \psi \land \theta
\]

(iii) We now obtain, from S and E, the following slice S':

\[
\psi \searrow \to z \leftarrow \hat{w}
\]

(iv) Call \( Q := \hat{w} r \to z \searrow \leftrightarrow \psi \) the slice under complement in E. So, slice S' is:

\[
\overline{Q} \searrow \to \hat{u} r \to v \searrow \leftrightarrow \theta
\]

We have a homomorphism \( \eta \) from Q to S given by \( w \mapsto u, z \mapsto v \):

\[
Q \hat{w} r \to z \searrow \leftrightarrow \psi
\]

\[
S \hat{u} r \to v \searrow \leftrightarrow \theta
\]

Slice S' has conflict at node u (with slice witness Q).

We now provide intuitive explanations for these steps, using R for the relation of r.

(i) The states pertaining to E are those not pertaining to slice \( \hat{w} r \to z \searrow \leftrightarrow \psi \).

The states s pertaining to this slice are those for which there is a state t such that \( s R t \) and t satisfies the unary arc \( z \searrow \leftrightarrow \psi \) (i.e. t satisfies formula \( \psi \)).

(ii) The states s pertaining to S are those for which there is a state t such that \( s R t \) and t satisfies both arcs \( v \searrow \leftrightarrow \psi \) and \( v \searrow \leftrightarrow \theta \) (i.e. t satisfies \( \psi \land \theta \)).

(iii) The states pertaining to slice S' are those pertaining to slice S that satisfy the unary arc \( u \searrow \leftrightarrow E \) (i.e. \( u \searrow \leftrightarrow \overline{Q} \)).

(iv) Any state pertaining to S must (as \( \eta : Q \to S \)) pertain to Q, whence it does not pertain to \( \overline{Q} \). Thus, there is no state pertaining to S', so it is not satisfiable.

Hence, set \{\( \langle r \rangle (\psi \land \theta), \langle r \rangle \psi \}\} is unsatisfiable and \( \langle r \rangle (\psi \land \theta) \models \langle r \rangle \psi \).
3 Graph Concepts and Results

We now introduce some basic concepts and results about graphs. We will use an infinite set \( \mathbb{N} \) of nodes; the first 3 nodes being \( x, y \) and \( z \).

A graph language \( \mathcal{G} \) is characterized by 2 sets of symbols: \( S_{b1} \), of unary ones, and \( S_{b2} \), of binary ones. Its syntax is defined by mutual recursion as follows.

(E) The expressions are the 1-ary symbols \( s \in S_{b1} \), the slices and the graphs (see below) and \( E \) (for an expression \( E \)).

(a) The arcs over a set \( N \subseteq \mathbb{N} \) are as follows.
   (1) A unary arc \( \langle w \rangle \) over \( N \) consists of a node \( w \in N \) and an expression \( E \).
   (2) A binary arc \( \langle u \, L \, v \rangle \) over \( N \) consists of nodes \( u, v \in N \) and 2-ary symbol \( L \in S_{b2} \).

(b) A sketch \( \Sigma = \langle N; A \rangle \) consists of 2 sets: \( N \subseteq \mathbb{N} \) (of nodes) and \( A \) of arcs over \( N \).

(D) A draft is a sketch with finite sets of nodes and arcs.

(g) A graph is a finite set of slices.

A proper sketch has non-empty node set. The positive part of a sketch consists of its nodes and its complement-free arcs. The empty graph \( \{ \} \) has no slices.

A structure \( \mathcal{M} \) for graph language \( \mathcal{G} \) consists of a universe \( M \neq \emptyset \) as well as a subset \( s_{b1} \subseteq M \) (for \( s \in S_{b1} \)) and a binary relation \( L_{b2} \) on \( M \) (for \( L \in S_{b2} \)).

We now define semantics also by mutual recursion.

(E) The extension \( [E]_{\mathcal{M}} \) of expression \( E \) is defined as follows. For a 1-ary symbol \( s \in S_{b1} \): \( [s]_{\mathcal{M}} = s_{b1} \); if \( E \) is a slice or a graph, we use its behaviour: \( [E]_{\mathcal{M}} = [E]_{\mathcal{M}} \) (see below); and \( [E]_{\mathcal{M}} = M \setminus [E]_{\mathcal{M}} \).

(f) An assignment for \( N \subseteq \mathbb{N} \) is a function \( g : N \rightarrow M \) (so \( w \in N \mapsto w^g \in M \)).

(a) We define satisfaction for an arc over set \( N \) as follows.
   (1) Assignment \( g \) satisfies unary arc \( \langle w \rangle \) iff \( w^g \in [E]_{\mathcal{M}} \).
   (2) Assignment \( g \) satisfies binary arc \( \langle u \, L \, v \rangle \) iff \( (u^g, v^g) \in L_{b2} \).

(c) Assignment \( g \) satisfies a sketch iff it satisfies all its arcs.

(5) For a slice \( S = \langle S ; w \rangle \), its behaviour \( [S]_{\mathcal{M}} \) consists of the values \( w^g \in M \) for the assignments \( g \) satisfying its underlying draft \( S \).

(g) For a graph \( G \), its behaviour is \( [G]_{\mathcal{M}} = \bigcup_{S \in G} [S]_{\mathcal{M}} \).

We define satisfiability, equivalence and nullity as follows. Consider a class of models \( \mathcal{K} \). A sketch \( \Sigma \) is satisfiable in \( \mathcal{K} \) iff there exist a model \( \mathcal{M} \in \mathcal{K} \) and an assignment satisfying \( \Sigma \) in \( \mathcal{M} \). A slice \( S \) is satisfiable in \( \mathcal{K} \) iff its underlying draft \( S \) is so; and a graph \( G \) is satisfiable in \( \mathcal{K} \) iff some slice \( S \in G \) is so. Expressions \( E \) and \( F \) are equivalent on \( \mathcal{K} \) iff, for every model \( \mathcal{M} \in \mathcal{K} \): \( [E]_{\mathcal{M}} = [F]_{\mathcal{M}} \). An expression \( E \) is null on \( \mathcal{K} \) iff, for every model \( \mathcal{M} \in \mathcal{K} \): \( \cap_{E \in \mathcal{K}} [E]_{\mathcal{M}} = \emptyset \). We use simply satisfiable, equivalent (noted \( \equiv \)) and null when referring to the class of all models. For instance, a singleton graph \( \{ S \} \) and its slice \( S \) are equivalent (so we can identify them); the empty graph \( \{ \} \) is null, as is the formula \( \bot \).

5 For more details about graphs see, e. g. [9, 10] and references therein.
We now define structural comparison and conflicts, introduced in Section 2.

For sketches \( \Delta \) and \( \Sigma \), a \textit{morphism} from \( \Delta \) to \( \Sigma \) is a function \( \mu : N_\Delta \to N_\Sigma \) (noted \( \mu : \Delta \to \Sigma \)), for which we have \( \mu(a) \in A_\Sigma \), for every arc \( a \in A_\Delta \) (with \( \mu(w|E) := w|E \) and \( \mu(u L v) := u L v^\mu \)). Now, given slices \( Q = (Q : v) \) and \( S = (S : u) \), a \textit{homomorphism} from \( Q \) to \( S \) is a function \( \eta : N_Q \to N_S \) (noted \( \eta : Q \to S \)) that is a morphism \( \eta : Q \to S \) and \( \eta v = u \).

Morphisms transfer satisfying assignments by composition. Given a morphism \( \mu : \Delta \to \Sigma \), if \( g \) satisfies \( \Sigma \) in \( \mathfrak{M} \), then the composite \( g \cdot \mu \) satisfies \( \Delta \) in \( \mathfrak{M} \). If there exists a homomorphism \( \eta : Q \to S \), then \( \mathfrak{M}[Q] \triangleright= \mathfrak{M}[S] \).

Consider a sketch \( \Sigma = (N; A) \). An expression \( E \) with \( u|E, u|F \in A \) is an \textit{expression witness} of \( \Sigma \) at node \( u \). A slice \( Q = (Q : v) \) for which \( v^\mu |Q \in A \) is a \textit{slice witness} of \( \Sigma \) at node \( v^\mu \in N \). A sketch is zero iff it has some witness. A slice \( S \) is zero iff \( S \) is zero. A graph is zero iff all its slices are zero. Clearly, a zero sketch is not satisfiable; so zero slices and graphs are null. One can effectively decide whether a draft, a slice or a graph is zero.

We use ‘+’ for adding arcs (and their nodes). To glue a slice \( Q \) on node \( w \) of slice \( S \), we take a copy \( \overline{Q} \) of \( Q \) having only its distinguished node \( w \) in common with \( S \) and add \( \overline{Q} \) to \( S \), thereby obtaining a \textit{glued slice} \( S_wQ \). We glue a graph by gluing its slices: \( S_wH = \{S_wQ / Q \in H\} \). For instance, \( S = \hat{\gamma} L w \rightarrow \gamma E \) and \( Q = \hat{\gamma} K v \rightarrow \gamma F \) have \( S_wQ = \hat{\gamma} L w \overset{K}{\longrightarrow} \gamma v \rightarrow \gamma F \) as glued slice.

A proper sketch \( \Sigma = (N; A) \) gives a \textit{natural structure} \( \mathfrak{N}[\Sigma] \): with universe \( N \), \( s^{\mathfrak{N}[\Sigma]} = \{w \in N / w|E \in A\} \) \((s \in S_1)\), \( L^{\mathfrak{N}[\Sigma]} = \{(u, v) \in N^2 / u L v \in A\} \) \((L \in S_2)\).

**Example 3.1** (Natural construction) Consider the following draft \( D \):

![Diagram of a draft](image)

The positive part \( D_+ \) of \( D \) and the natural structure \( \mathfrak{N}[D] \) are as follows:

![Diagram of the natural structure](image)

Consider the identity assignment \( 1 \) on set \( \{x, y\} \).

(i) Assignment \( 1 \) satisfies the arcs of \( D_+ \) as well as the 1-ary arc \( x|\overline{s} \).

(ii) We can see that assignment \( 1 \) also satisfies the unary arc:

![Diagram of the unary arc](image)
Thus, assignment 1 satisfies draft D in natural structure N[D].

In the sequel, we will show how one can represent a modal formula by an expression (of a graph language) with the same meaning, thus reducing questions about formulas to questions on expressions. We will be able to eliminate logical symbols from a modal formula, converting it to an equivalent expression. The elimination rules mimic the semantics of the modal language. The following ones are general.

We can eliminate double complementation: \( \overline{E} \leftrightarrow 1 \overline{E} \). We can also move complementation inwards by rules akin to De Morgan laws: rule (\( \cup \)) converts a complemented graph \( \overline{G} \) to the slice having 1-ary arcs \( \overline{w} \rightarrow \varepsilon \), rule (\( \cap \)) converts a slice \( S \) to a graph with slices \( \overline{w} \rightarrow \varepsilon \). So, \( \{ \} \leftrightarrow \overline{x} \), \( \overline{w} \rightarrow \varepsilon \leftrightarrow \{ \} \) and \( \overline{w} \rightarrow \varepsilon \leftrightarrow \{ \} \).

Structural rules lower graphs and slices occurring within 1-ary arcs. Rule (\( \leftarrow \lor \)) converts slice \( S + w \rightarrow \varepsilon \) to the graph \( \{ S + w \rightarrow \varepsilon / Q \in H \} \) and rule (\( \leftarrow \land \)) converts slice \( S + w \rightarrow \varepsilon \) to the glued slice \( S w \uparrow \). We thus have a derived rule (\( \leftarrow \)) converting slice \( S + w \rightarrow \varepsilon \) to the glued graph \( S w \). So, \( S + w \rightarrow \varepsilon \leftrightarrow \{ \} \).

The zero rule (\( Z \)) erases a slice with conflict. The alternative expansion rule (\( w | E \)) expands a slice \( S \) to graph \( \{ S + w \rightarrow \varepsilon, S + w \rightarrow \varepsilon \} \). From (\( w | E \)) and (\( Z \)) we can derive the shift expansion rule: given a slice \( S \) with 1-ary arc \( u \rightarrow v \rightarrow w \rightarrow \varepsilon \) and 2-ary arc \( u \rightarrow v \rightarrow w \rightarrow \varepsilon \), \( S \) expands to

\[
S + u \rightarrow v \rightarrow w \rightarrow \varepsilon .
\]

The expression-set rule converts set \( \{ E_1, \ldots, E_n \} \) to the slice \( E_1 \ldots E_n \).

We also have rules aiming to capture special properties of a relation.

(\( R[L] \)) For a reflexive relation \( L \): expand node \( w \) to \( w \uparrow \).

(\( T[L] \)) For a transitive relation \( L \): expand \( u \rightarrow v \rightarrow w \rightarrow \varepsilon \) to \( u \rightarrow v \rightarrow w \rightarrow \varepsilon \).

(\( S[L] \)) For a symmetric relation \( L \): expand \( u \rightarrow v \rightarrow \varepsilon \) to \( u \rightarrow v \rightarrow \varepsilon \).

(\( A[L] \)) For an anti-symmetric relation \( L \): identify nodes \( u \) and \( v \) with \( u \rightarrow v \rightarrow \varepsilon \).

We often use \( u \rightarrow v \rightarrow \varepsilon \) as short for \( u \rightarrow v \rightarrow \varepsilon \).

A derivation is a finite sequence of rule applications. We use \( E \vdash F \) for \( E \) derives \( F \). The graph calculus is refutationally sound and complete: a finite expression set

\[6\] This shift expansion can be used to simulate the modal \( [ ] \) transfer \[6\].

122
\( \mathcal{E} \) is null iff \( \mathcal{E} \) derives the empty graph \( (\mathcal{E} \vdash \{ \} ) \). It is also sound for equivalence: if \( \mathcal{E} \) derives \( F \) \( (\mathcal{E} \vdash F) \), then \( \mathcal{E} \equiv F \). [9,10]

4 Intuitionistic Modal Logic: Flat Semantics

We now examine flat semantics for intuitionistic modal logic [8]. It is convenient to consider a hierarchy of structures.

A pre-relational structure \( \mathcal{B} \) consists of a set \( W \neq \emptyset \) (of worlds) with a special binary relation \( \leq \) on \( W \), together with a binary relation \( R \) on \( W \) (for \( r \in \mathcal{R} \)) and a valuation \( \mathcal{V} : \mathcal{P}L \to \mathcal{P}(W) \). We introduce predicate extension \( \mathcal{P} := \{ u \in W / p \in \mathcal{M}(u) \} \) (for \( p \in \mathcal{P}L \)). We use the abbreviations \( \mathcal{P}^0 \) and \( \mathcal{R} \) for \( r^0 \).

Formula satisfaction (with \( \mathcal{I} \) as short for \( \mathcal{I} - \mathcal{B} \)) is as follows. The local cases are:

\[
\begin{align*}
\mathcal{I} \vdash \bot & \iff u \notin \mathcal{V}(p) \quad \text{(i.e. } u \in \mathcal{P}) \quad \mathcal{I} \vdash (\psi \land \theta) & \iff \mathcal{I} \vdash \psi \quad \text{and} \quad \mathcal{I} \vdash \theta \quad \mathcal{I} \vdash (\psi \lor \theta) & \iff \mathcal{I} \vdash \psi \quad \text{or} \quad \mathcal{I} \vdash \theta. \\
\mathcal{I} \vdash \neg \psi & \iff, \text{ for every } v \in W, v \mathcal{R} w \text{ and } \mathcal{I} \vdash \psi, \\
\mathcal{I} \vdash \psi \rightarrow \theta & \iff, \text{ for every } v \geq u, v \mathcal{I} \vdash \theta \quad \text{(i.e. there exists no } v \geq u \text{ such that } v \mathcal{I} \vdash \psi). \
\end{align*}
\]

A relational structure is a pre-relational structure \( \mathcal{B} \) where relation \( \leq \) is a partial order on \( W \). To have monotonicity of satisfaction, one restricts relational structures to birelational structures by imposing 3 extra requirements. Monotone valuation:

\[
\begin{align*}
\forall u \leq u', \mathcal{V}(u) \subseteq \mathcal{V}(u') \quad \text{(F1): given } u', u, v \in W, \text{ such that } u' \geq u \text{ and } u \mathcal{R} v, \text{ there exists } v' \in W, \text{ such that } u' \mathcal{R} v' \text{ and } v' \leq v'. \\
\forall u, v, v' \in W, \text{ such that } u \mathcal{R} v \text{ and } v \leq v', \text{ there exists } u' \in W, \text{ such that } u \leq u' \text{ and } u' \mathcal{R} v'. \quad \text{[8, p. 50]}
\end{align*}
\]

We wish to reason graphically about flat semantics with a symbol \( \leq \). For this purpose we consider a graph language \( \mathcal{G} \) with \( \mathcal{S}_1 = \Phi \) and \( \mathcal{S}_2 = \mathcal{R} \cup \{w\} \). We draw w-arrows as \( \longrightarrow \). In a structure \( \mathcal{M} \) for \( \mathcal{G} \), we set \( \varphi^{\mathcal{M}} := \{ u \in W / u \mathcal{I} - \mathcal{M} \varphi \} \). Of course, it suffices to give \( \mathcal{P}^{\mathcal{M}} \), for \( p \in \mathcal{P}L \).

Then, we can handle logical symbols by the 7 pre-relational elimination rules converting formulas to equivalent expressions as shown in Table 1. These rules transcribe formula satisfaction in graph terms, which guarantees their soundness. For instance, for \( (\neg) \), we have: \( u \notin \neg \varphi^{\mathcal{M}} \iff u \mathcal{I} - \mathcal{M} \neg \varphi \) iff there is some \( v \geq u \) such that \( v \mathcal{I} \varphi \) iff assignment \( g \) with \( x^g = u \) and \( y^g = v \) satisfies draft \( \vdash x \quad y \quad - \neg \varphi \)

\[
\begin{align*}
\text{iff } u \in [ \quad \text{iff } u \notin [ \quad \text{iff } u \notin [ \quad \text{iff } u \notin [ \\
\end{align*}
\]

One could also consider some variations. For the condition \( w \mathcal{I} - \mathcal{M} \langle r \rangle \varphi \), there are \( w_0, v_0 \in W \) such that \( w \geq w_0 \), \( w_0 \mathcal{R} v_0 \) and \( v_0 \mathcal{I} \varphi \) (attributed to Plotkin and Stirling [8, p. 49]), we obtain the slice \( \vdash x \quad y \quad z \quad - \neg \varphi \). We could similarly handle a condition like \( w \mathcal{I} - \mathcal{M} \langle r \rangle \varphi \), for all \( w' \in W \), if \( w' \geq w \) there exists \( v' \in W \), such that \( w' \mathcal{R} v' \) and \( v' \mathcal{I} - \mathcal{M} \varphi \).
Table 1
Pre-relational elimination rules

<table>
<thead>
<tr>
<th>Formula</th>
<th>≈</th>
<th>Expression</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥</td>
<td>(⊥)</td>
<td>{ }</td>
<td>empty graph</td>
</tr>
<tr>
<td>ψ ∧ θ</td>
<td>(∧)</td>
<td>ψ ↝ ¬ ¬ ψ ¬ ¬ θ</td>
<td>single-node slice</td>
</tr>
<tr>
<td>ψ ∨ θ</td>
<td>(∨)</td>
<td>{ \ψ ↝ ¬ ¬ ψ , } \ψ ↝ ¬ ¬ θ }</td>
<td>graph with single-node slices</td>
</tr>
<tr>
<td>⟨r⟩ ϕ</td>
<td>(⟨⟩)</td>
<td>(r) ↝ ϕ</td>
<td>2-node slice</td>
</tr>
<tr>
<td>¬ ϕ</td>
<td>(¬)</td>
<td>¬ ϕ</td>
<td>complemented 2-node slice</td>
</tr>
<tr>
<td>ψ → θ</td>
<td>(→)</td>
<td>ψ ↝ ϕ</td>
<td>complemented 2-node slice</td>
</tr>
<tr>
<td>[r] ϕ</td>
<td>([)</td>
<td>(r) ↝ ϕ</td>
<td>complemented 3-node slice</td>
</tr>
</tbody>
</table>

The next result illustrates how one can obtain expressions for complex formulas from those of its immediate sub-formulas.

Proposition 4.1 (Derived pre-relational conversions) The pre-relational conversions in Tables 2 and 3 are derived.

Proof. By conversion, structural and complement rules. \(\square\)

For instance, for (⟨⟩): ⟨r⟩ ϕ ≈ \(r\) ↝ ϕ; for (→): ψ → θ ≈ ψ ↝ ϕ; for (∪): [r] (ψ ∧ θ) ≈ [r] ψ ∧ θ.
Table 2
Derived pre-relational slice conversions (the nodes with '*' are new)

\[
\begin{align*}
(\bot) \quad & S + w - - \bot \approx^* \{ \} \\
(\land) \quad & S + w - - \psi \land \theta \approx^* S + w - - \psi \quad S + w - - \psi \land \theta \approx^* \{ S + w - - \bar{\psi}, \} \\
(\lor) \quad & S + w - - \psi \lor \theta \approx^* \{ S + w - - \bar{\psi}, \} \quad S + w - - \psi \lor \theta \approx^* S + w - - \bar{\psi} \\
(\neg) \quad & S + u - - <\neg \varphi> \approx^* S + u \quad S + u - - \psi \rightarrow \theta \approx^* S + u \psi \\
(\langle\rangle) \quad & S + u - - \langle r \rangle \varphi \approx^* \langle r \rangle \quad S + u - - \langle r \rangle \psi \land \theta \approx^* S + u \psi \\
([\langle\rangle]) \quad & [\langle r \rangle] \varphi \approx^* [\langle r \rangle] \quad [\langle r \rangle] \psi \land \theta \approx^* [\langle r \rangle] \psi \\
([\langle\rangle] \land) \quad & [\langle r \rangle] \land \psi \theta \approx^* [\langle r \rangle] \land \psi \theta \\
\end{align*}
\]

Table 3
Derived pre-relational expression conversions

\[
\begin{align*}
(\neg\neg) \quad & \neg\neg \varphi \approx^* \hat{x} \quad \hat{x} \rightarrow \rightarrow \rightarrow \rightarrow y \\
(\neg\neg\langle\rangle) \quad & \neg\neg \langle r \rangle \varphi \approx^* \hat{x} \quad \hat{x} \rightarrow \rightarrow \rightarrow \rightarrow y \\
(\neg\neg([\langle\rangle])) \quad & \neg\neg([\langle r \rangle]) \varphi \approx^* \hat{x} \quad \hat{x} \rightarrow \rightarrow \rightarrow \rightarrow y \\
(\neg\neg([\langle\rangle] \land)) \quad & \neg\neg([\langle r \rangle] \land \psi \theta) \approx^* \hat{x} \quad \hat{x} \rightarrow \rightarrow \rightarrow \rightarrow y \\
\end{align*}
\]
Example 4.2 (Pre-relational consequence) To show that \([r] \psi\) is a consequence of \([r] (\psi \land \theta)\), we first convert \([r] (\psi \land \theta) \approx^* S'\) and \([r] \psi \approx^* S''\) (cf. Table 3). We then have:

\[
\begin{align*}
[r] (\psi \land \theta) & \quad S' \quad \approx^* \quad S'' \quad (\approx^2) \quad \vdash \quad \psi' \quad \dashv \quad \phi' \\
[r] \psi & \quad \vdash \quad \psi \\
\end{align*}
\]

The resulting slice \(S\) is zero: slice \(S\) has as witness at node \(x\) the slice \(\overline{\psi} \dashv \vdash \overline{\psi} \vdash \overline{\psi} \vdash \overline{\psi}\) under morphism \(x \rightarrow x, y \rightarrow y\). (Notice that \(S\) can be shifted to a slice with expression witness \(\psi \) at \(x\).) Thus \(S\) is unsatisfiable. Hence, set \(\{ [r] (\psi \land \theta), [r] \psi \}\) cannot be satisfied in a pre-relational structure.

We now indicate how our calculus handles the necessitation rule: “from theorem \(\psi\), infer \([r] \psi\)”. Its refutational analogue is: “if \(\overline{\psi} \vdash \{ \}\), then \([r] \psi \vdash \{ \}\)”. We have:

\[
[r] \varphi \approx^* \hat{x} \quad y \quad r \quad z \quad \overline{\psi} \vdash \hat{x} \quad y \quad r \quad z \quad \vdash \{ \} \approx^* \{ \}. 
\]

We also have rules coming from the intended meaning of \(w_c\) as \(\leq\).

The relational operational rules are \((\text{Ref}[wc]), (\text{As}[wc])\) and \((\text{Tr}[wc])\) (cf. Section 3).

A sketch is \(w_c\)-reduced iff \(u = v\), whenever it has arcs \(u \leftrightarrow v\). Every draft can be contracted to a \(w_c\)-reduced draft.\(^8\)

For birelational structures, we also have the 3 birelational transformation rules:

- \((p)\) Contract slice \(S + p \vdash \vdash u \quad u' \quad \overline{\neg p}\) to the empty graph \(\{ \}\).
- \((F1)\) Expand slice \(S + u' \quad u \quad r \quad v \quad S + u' \quad u \quad r \quad v\) (with new \(v^*\)).
- \((F2)\) Expand slice \(S + u \quad r \quad v \quad v' \quad S + u \quad r \quad v \quad v'\) (with new \(u^*\)).

Then, we can derive the following birelational formula transfer: \(S + \varphi \vdash \vdash u \quad u' \approx^* S + \varphi \vdash \vdash u \quad u' \vdash \neg \varphi\). (By the alternative expansion rule \((w'[E])\): case \((i)\) \(\varphi\) follows from \((F1)\) and case \([r] \varphi\) follows from \((\text{Tr}[wc])\)).

Example 4.3 (Birelational consequence) To show that \(\neg \neg \varphi\) is a birelational consequence of \(\varphi\), we consider the set \(\{ \varphi, \neg \neg \varphi \}\).

1. \(\{ \varphi, \neg \neg \varphi \} \approx^* S_1\), with \(S_1 = \varphi \vdash \vdash \hat{x} \quad z \quad \overline{\neg \varphi}\).

\(^8\) For instance, \(x \quad y \quad \overline{\neg p}\) is not \(w_c\)-reduced, but it contracts to the \(w_c\)-reduced \(z \quad \overline{\neg p}\).
(ii) By reflexivity ($R_w$), slice $S_1$ expands to the following slice $S_2$:

\[
\varphi \vdash x \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \xrightarrow{\phantom{\vec{\rightarrow}}} y \xleftarrow{\phantom{\vec{\rightarrow}}} \varphi \]

(iii) Transfer formula $\varphi$ from $x$ to $z$ to obtain the following slice $S_3$:

\[
\varphi \vdash x \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \xrightarrow{\phantom{\vec{\rightarrow}}} y \xleftarrow{\phantom{\vec{\rightarrow}}} \varphi \]

Slice $S_3$ is zero: slice $S_3$ has as witness at node $z$ the slice $\vec{x} \xrightarrow{\phantom{\vec{\rightarrow}}} y \leftarrow \varphi$ under morphism $x, y \mapsto z$. So, $S_3$ is unsatisfiable.

Hence, set $\{\varphi, \neg \neg \varphi\}$ cannot be satisfied in a birelational structure.

We can show graphically that the following formulas (cf. [8, p. 51, 52]) are birelationally valid: $\langle r \rangle \psi \rightarrow \theta \rightarrow \langle r \rangle \theta$, $\langle r \rangle \psi \rightarrow \theta \rightarrow \langle r \rangle \theta$, $\neg \langle r \rangle \bot$, $\langle r \rangle \psi \lor \theta \rightarrow \langle r \rangle \psi \lor \langle r \rangle \theta$, $\langle r \rangle \psi \rightarrow \langle r \rangle \theta$. (In fact, $\neg \langle r \rangle \bot$ can be seen to be pre-relationally valid.)

The natural construction (in Section 3) applied to a proper $w_c$-reduced $G_{\mathcal{L}}$-sketch gives a pre-relational structure.

**Example 4.4** (Birelational non-consequence) To show that $p$ is not a birelational consequence of $\neg \neg p$, we consider the set $\{\neg \neg p, p\}$.

(i) Set $\{\neg \neg p, p\}$ converts to the following slice $S_1$:

\[
\neg \neg p \vdash \vec{x} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \vec{x} y \xleftarrow{\phantom{\vec{\rightarrow}}} \neg \neg p
\]

(ii) By reflexivity ($\mathbb{R}_w$), slice $S_1$ expands to the following slice $S_2$:

\[
\neg \neg p \vdash \vec{x} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \vec{x} y \xleftarrow{\phantom{\vec{\rightarrow}}} \neg \neg p
\]

(iii) Now, slice $S_2$ shifts (cf. Section 3) to the following slice $S_3$:

\[
\neg \neg p \vdash \vec{x} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \vec{x} y \xleftarrow{\phantom{\vec{\rightarrow}}} \neg \neg p
\]

(iv) By eliminating double complement and lowering, we obtain slice $S_4$:

\[
\neg \neg p \vdash \vec{x} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \vec{x} y \xleftarrow{\phantom{\vec{\rightarrow}}} \neg \neg p
\]

\[\hat{\varphi} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \vec{x} y \xleftarrow{\phantom{\vec{\rightarrow}}} \neg \neg p\]

\[\hat{\varphi} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \vec{x} y \xleftarrow{\phantom{\vec{\rightarrow}}} \neg \neg p\]

\[\hat{\varphi} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \vec{x} y \xleftarrow{\phantom{\vec{\rightarrow}}} \neg \neg p\]

\[\hat{\varphi} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \vec{x} y \xleftarrow{\phantom{\vec{\rightarrow}}} \neg \neg p\]

\[\hat{\varphi} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \xrightarrow{\phantom{\vec{\rightarrow}}} \vec{\rightarrow} \vec{x} y \xleftarrow{\phantom{\vec{\rightarrow}}} \neg \neg p\]

9 Notice that $S_3$ can be shifted to a slice with expression witness $\varphi$ at $z$. 

127
(v) The positive part of $S_4$ and corresponding natural structure $B$ are:

```
\[
\begin{array}{c}
\begin{array}{c}
\hat{x} \\
\hat{y} \\
\end{array}
\end{array}
\begin{array}{c}
\hat{x} \\
\hat{y} \\
\end{array}
\end{array}
\]
```

Note that structure $B$ is birelational. Much as in Example 3.1 (Natural construction), we see that the identity assignment $1$ satisfies draft $S_4$ in $B$.

Hence, slice $S_1$ and set $\{¬¬p, p\}$ are satisfiable in a birelational structure. Thus, $p$ is not a birelational consequence of $¬¬p$.

The special binary relation $≤$ of a pre-relational structure may be symmetric. For such cases, we use the rule ($R_m[wc]$) (cf. Section 3).

**Example 4.5** (Symmetric birelational consequence) To show that $φ$ is a symmetric birelational consequence of $¬¬φ$, we consider the set $\{¬¬φ, φ\}$.

(i) Set $\{¬¬φ, φ\}$ converts to the following slice $S_1$:

```
\[
\begin{array}{c}
\begin{array}{c}
\hat{x} \\
\hat{y} \\
\end{array}
\end{array}
\begin{array}{c}
\hat{x} \\
\hat{y} \\
\end{array}
\end{array}
\]
```

(ii) By ($R_f[wc]$) and graph rules, we can transform $S_1$ to the following slice $S_2$:

```
\[
\begin{array}{c}
\begin{array}{c}
\hat{x} \\
\hat{y} \\
\end{array}
\end{array}
\begin{array}{c}
\hat{x} \\
\hat{y} \\
\end{array}
\end{array}
\]
```

(iii) By symmetry ($R_m[wc]$), we can expand $S_2$ to the following slice $S_3$:

```
\[
\begin{array}{c}
\begin{array}{c}
\hat{x} \\
\hat{y} \\
\end{array}
\end{array}
\begin{array}{c}
\hat{x} \\
\hat{y} \\
\end{array}
\end{array}
\]
```

(iv) We can transfer formula $φ$ from $y$ to $x$, expanding $S_3$ to the slice $S_4$:

```
\[
\begin{array}{c}
\begin{array}{c}
\hat{x} \\
\hat{y} \\
\end{array}
\end{array}
\begin{array}{c}
\hat{x} \\
\hat{y} \\
\end{array}
\end{array}
\]
```

This slice $S_4$ has a conflict at node $x$ (with formula $φ$ as expression witness). So, in a birelational structure with symmetric $≤$, one cannot satisfy $\{¬¬φ, φ\}$.

We can similarly show that $φ ∨ ¬φ$ is valid in symmetric birelational structures.
5 Intuitionistic Modal Logic: Decoupled Semantics

We now examine another semantics for intuitionistic modal logic.

The basic motivation comes from decoupling objects and stages [4]. A stratified structure consists of a set I (of stages) partially ordered by ≤ with, for each i ∈ I: a universe Ai ≠ ∅ (of objects), a subset Pi ⊆ Ai (for p ∈ PL) and binary relation R_i on A_i (for r ∈ RS). We use the abbreviations: P_i for P_i^3 and R_i for R_i^3.

We prefer another formulation as follows. A pre-graded structure \( \mathfrak{A} \) consists of 2 sets \( I \), with a special binary relation ≤ on it, and \( A \neq \emptyset \), it has as domain a non-empty subset \( A \times I \) and (with abbreviations \( P \) for \( P^3 \) and \( R \) for \( R^3 \)) a subset \( P \subseteq A \times I \) (for p ∈ PL) and 2-ary relation \( R \) on \( A \times I \) (for r ∈ RS) such that \( i = j \) whenever \( \langle a, i \rangle R \langle b, j \rangle \). We can introduce a special relation on ordered pairs by \( \langle a, i \rangle \leq \langle b, j \rangle \) iff \( i \leq j \). We then obtain a pre-relational structure.

Satisfaction (with \( \models \) as short for \( \models_{\mathfrak{A}} \)) is as follows. For \( \bot, \land \lor \text{ and } (\cdot, ) \), it is as in Section 4 with \( u = \langle a, i \rangle \). For \( \vdash \langle a, i \rangle \models \neg \varphi \) iff, for every \( j \geq i \), \( \langle a, j \rangle \not\models \varphi \) (i.e. there exists no \( j \geq i \) such that \( \langle a, j \rangle \models \varphi \)). For \( \rightarrow \langle a, i \rangle \models \psi \rightarrow \theta \) iff, for every \( j \geq i \), if \( \langle a, j \rangle \models \psi \) then \( \langle a, j \rangle \models \theta \) (i.e. there exists no \( j \geq i \) such that \( \langle a, j \rangle \not\models \psi \) and \( \langle a, j \rangle \not\models \theta \)). For \( (\cdot) \langle a, i \rangle \models [r] \varphi \) iff, for all \( j \geq i \) and \( b \in A \). if \( \langle a, j \rangle R \langle b, j \rangle \) then \( \langle b, j \rangle \models \varphi \) (i.e. there exist no \( j \geq i \) and \( b \in A \) such that \( \langle a, j \rangle R \langle b, j \rangle \) and \( \langle b, j \rangle \not\models \varphi \)).

As in Section 4, we consider some restrictions. A graded structure is a pre-graded structure \( \mathfrak{A} \) where special relation ≤ is a partial order on \( I \). A growing graded structure is one with growing universes, predicates and relations. For \( i \leq j \in I \): if \( \langle a, i \rangle \in A_i \) then \( \langle a, j \rangle \in A_i \) (i.e. \( A_i \subseteq A_j \); if \( \langle a, i \rangle \in P_i \) then \( \langle a, j \rangle \in P_j \) (i.e. \( P_i \subseteq P_j \)); if \( \langle a, i \rangle R \langle b, i \rangle \) then \( \langle a, j \rangle R \langle b, j \rangle \) (i.e. \( R_i \subseteq R_j \). On a growing graded structure, satisfaction is monotonic.

We wish to reason graphically about decoupled semantics with symbols \( \circ \) (for ≤) and \( \circ \text{ and } (\cdot) \) (with intended meaning \( \langle a, i \rangle \circ \langle b, j \rangle \) iff \( a = b \)). For this purpose we consider a graph language \( \mathfrak{Q} \) with \( S_1 = \Phi \) and \( S_2 = I \cup \{ \circ, (\cdot) \} \). We draw \( \circ \)-arrows as \( \circ \rightarrow \) and \( (\cdot) \)-arrows as \( \circ \rightarrow \circ \). In a structure \( \mathfrak{M} \) for \( \mathfrak{Q} \), as before, we set \( \varphi^{\mathfrak{M}} = \{ u \in M / u \models_{\mathfrak{M}} \varphi \} \) and it suffices to give \( P^\mathfrak{M} \), for \( p \in PL \).

Then, we can handle logical symbols by the 7 pre-graded elimination rules converting formulas to equivalent expressions, much as before. The rules for \( \bot, \land \lor \text{ and } (\cdot, ) \) are as in Table 1. The other 3 rules convert formulas \( \neg \varphi, \psi \rightarrow \theta \) and \( [r] \varphi \), respectively, to the following expressions:

Thus, we have derived pre-graded conversions much as those in Proposition 4.1.

We also have operational rules coming from the intended meanings of \( \circ \text{ and } (\cdot) \). For \( \circ \), we have \( (R[\circ]) \), \( (S[\circ]) \) and \( (T[\circ]) \). For graded \( \circ \), we have \( (R[x]) \), \( (S[x]) \) and \( (T[x]) \), as well as the rule identifying nodes \( u \) and \( v \) such that \( u \xleftarrow{r} v \). For a symmetric ≤, we use \( (S[x]) \).

\footnote{Note that these restrictions are simpler and more intuitive than requirements (F1) and (F2).}
For growing graded structures, we also have the following 3 growing transformation rules. For domain: given \( u \sim v \), add \( u \leadsto u^* \sim v \) (with new node \( u^* \)). For \( p \in \mathcal{PL} \): erase slice with \( p \sim u \sim v \sim \neg \neg p \). For \( r \in \mathcal{RS} \): given \( u' \sim v \sim v' \), add \( u' \rightarrow v' \). Then, we can derive the growing formula transfer: \( S + \varphi \sim u \sim v \Rightarrow S + \varphi \sim u \sim v \sim \neg \neg \varphi \).

We can establish consequence as in Section 4 with \( i \in c \) and \( e \in o \) in lieu of \( w \in c \). We can establish non-consequence much as in Section 4, even though the natural construction is now more involved. A sketch is \( \Sigma \) = sketch \( \Sigma = \langle N; A \rangle \) gives a growing-graded consequence of \( \varphi \) as in Example 4.3, and that \( \varphi \) is a symmetric growing-graded consequence of \( \neg \neg \varphi \) as in Example 4.5 (note that symmetric growing graded structures have constant universes, predicates and relations).

We can also establish non-consequence much as in Section 4, even though the natural construction is now more involved. A sketch is \( \cap \)-reduced iff \( u = v \), whenever it has arcs \( u \sim v \). The natural construction applied to a proper \( \cap \)-reduced \( \mathcal{G}' \)-sketch \( \Sigma = \langle N; A \rangle \) gives a pre-graded structure \( \mathfrak{A}[\Sigma] \) and assignment \( h_{\Sigma} \) as follows.

(i) Define 2-ary relations on \( N \): \( u \sim v \) iff \( u \sim v \in A \) and \( u \sim v \) iff \( u \sim v \in A \).

(ii) Define domain \( A_N = \{ [w]_w, [w]_w \in A \times I / w \in N \} \) and relation \( \leq \) on \( I \) by \( [u]_w \leq [v]_w \) iff \( u \sim v \in A \).

(iii) Define subsets by \( \langle [w]_w, [w]_w \rangle \in \mathcal{P} \) iff \( w \rightarrow p \in A \) and relations by \( \langle [w]_w, [w]_w \rangle \Rightarrow \langle [v]_v, [v]_v \rangle \) iff \( u \rightarrow v \in A \).

(iv) Define natural assignment \( h_{\Sigma} : N \rightarrow A_N \) by \( w \rightarrow \langle [w]_w, [w]_w \rangle \).

To see that \( p \) is not a growing-graded consequence of \( \neg \neg p \), we proceed as in Example 4.4 with \( i \in c \) and \( e \in o \) in lieu of \( w \in c \), and \( (\mathcal{G}[\Sigma]) \), \( (\mathcal{G}[\alpha]) \) and \( (\mathcal{G}[\alpha]) \). We obtain the final slice \( S' \).

The positive part \( D \) of \( S' \) and corresponding natural structure \( \mathfrak{A} \) are:

Note that structure \( \mathfrak{A} \) is growing graded.

We can see that the natural assignment \( h \) satisfies draft \( D \) in structure \( \mathfrak{A} \):

130
Much as before, we can see that the natural assignment $h$ satisfies draft $S_4'$ in structure $\mathfrak{A}$. Thus, slice $S_4'$ and set $\{\neg\neg p, p\}$ are satisfiable in a growing graded structure. Hence, $p$ is not a growing-graded consequence of $\neg\neg p$.

6 Extension to Multi-modal Logics

We have sound and complete graph calculi for flat and decoupled intuitionistic modal logics. We now indicate how to extend these calculi to multi-modal logics.

We can also allow some connections as well as some operations on relations (much as in [2]). For instance, we can express inclusion of relations by a rule $(L \subseteq K)$ adding $u \xrightarrow{K} v$ whenever we have $u \xrightarrow{L} v$ and intersection of relations by a rule $(L \cap K)$ adding $u \xrightarrow{L \cap K} v$. We can similarly express composition (by consecutive arrows), transposal (by arrow reversal) and identity (by node identification) [9,10]. For a set $\Delta$ of constraints, a $\Delta$-derivation is a finite sequence of rule applications and constraints in set $\Delta$.

Consider relation symbols $r, s', s''$ and $t$, subject to the constraints: “$r \subseteq s' \cap s''$, $s' \subseteq t$ and $t$ is transitive”. We construct a graph calculus by adding to our basic rules the set $\Delta$ consisting of the rules $(r \subseteq s' \cap s'')$, $(s' \subseteq t)$, $(s' \cap s'')$ and $(T \{t\})$.

(+) We can show that $\langle t \rangle \varphi$ is a $\Delta$-consequence of $\langle r \rangle \langle s' \rangle \varphi$, much as before: we transform set $\{\langle r \rangle \langle s' \rangle \varphi, \langle t \rangle \varphi\}$ to the following slice:

(−) We can also obtain a $\Delta$-model for $\{\langle r \rangle p, \langle t \rangle \langle t \rangle p\}$, much as before. We transform this set to a slice $S$, which gives a model $\mathfrak{A}$, as follows:
Now, consider graph languages $G_L$ (cf. Section 4) and $G_L'$ (cf. Section 5). We say that expressions $E$ of $G_L$ and $E'$ of $G_L'$ are associated (noted $E \simeq E'$) iff $E'$ is the result of replacing everywhere in $E$ $\iota_c$ by $\phi$ and $\phi_o$, and similarly for constraints. For sets of constraints $\Delta$ in $G_L$ and $\Delta'$ in $G_L'$, $\Delta \simeq \Delta'$ iff each $\delta \in \Delta$ has some associated $\delta' \in \Delta'$ and vice-versa. We call derivations $E_1, \ldots, E_n$ and $E'_1, \ldots, E'_n$ associated iff $E_i \simeq E'_i$, for $i = 1, \ldots, n$. Call an expression $E'$ of $G_L'$ neat iff $\iota_c$ and $\phi_o$ occur only in parallel arcs, and similarly for (sets of) constraints and derivations.

By $\leq \mapsto \iota_c$, $\phi_o$ we transform flat rules to decoupled ones and vice-versa. So, given associated constraint sets $\Delta \simeq \Delta'$, every flat $\Delta$-derivation $\Pi$ has an associated neat decoupled $\Delta'$-derivation $\Pi'$ (which will be graded or growing whenever $\Pi$ is relational or birelational) and, similarly, every neat decoupled $\Delta'$-derivation $\Pi'$ has an associated flat $\Delta$-derivation $\Pi$. Hence, a modal formula is flat-derivable iff it is decoupled-derivable. Thus, by completeness, the same modal formulas hold in flat (birelational) and decoupled (growing) structures.

7 Concluding Remarks

We have presented a flexible and uniform formalism for intuitionistic modal logics where one can express, analyse and compare possible-world semantics. Our approach explores the flexibility of graph calculi to express directly and graphically Kripke-based semantics of intuitionistic modal logics.

We have illustrated these ideas by applying them to 2 semantics in Sections 4 and 5 and indicated their extension to multi-modal logics in Section 6. Our approach is uniform: once we have expressed the semantics (including connections among relations), we apply the general (sound and complete) graph-calculus machinery. For flat and decoupled semantics, we have transcribed their satisfaction conditions graphically to expressions and used this to show that they give equivalent semantics. We have also illustrated (in Section 4) how one can express simple variations of the satisfaction conditions, which give different semantics on relational structures, though some of them may coincide on birelational structures.

We thus have a flexible, uniform, rigorous and intuitive formalism for visual exploration of intuitionistic modal logics.

References

A logical framework for multi-agent visual-epistemic reasoning

Valentin Goranko 1

Department of Philosophy, Stockholm University, Sweden

Abstract

We study a logical framework for multi-agent epistemic reasoning based on processing of visual information. This framework is modelled here by multi-agent systems where each agent receives visual information from the environment via mobile camera with a given angle of vision in the plane. The agents can thus observe their surroundings and each other and can reason about each other’s observation abilities and knowledge derived from these observations. We introduce suitable logical languages for describing such scenarios and formalising such reasoning, involving atomic formulae stating what agents can see, multi-agent epistemic operators for individual, distributed and common knowledge, as well as dynamic operators reflecting the ability of agents (or, their cameras) to move and turn around in order to reach positions satisfying formally specified visual-epistemic requirements. We introduce several different models for these languages and develop algorithmic methods for automated reasoning in our basic logical system, called ‘Big Brother Logic’ (BBL), and some natural extensions of it. In particular, we study the interaction between observational abilities and knowledge, both of which essentially depend on the underlying geometric constraints and assumptions. Besides being of purely logical interest, this work has potential applications to formal specification, verification and automated reasoning in multi-robot systems.

This talk is based on a revised and substantially extended version of the joint work [1] with Olivier Gasquet (Univ. Paul Sabatier, IRIT, Toulouse) and Francois Schwarzentruber (ENS Rennes).

Keywords: Multi-agent systems, mobile cameras, logic, visual-epistemic reasoning, model-checking

References


1 Email: valentin.goranko@philosophy.su.se
Classical resolution for many-valued logics

João Marcos
LoLITA & Department of Informatics and Applied Mathematics, UFRN, Brazil

Cláudia Nalon
Departament of Computer Science, University of Brasília, Brazil

Abstract
We present a resolution-based proof method for finite-valued propositional logics based on an algorithmic reduction procedure that expresses these logics in terms of bivalent semantics. Our approach is hybrid in using some elements which are internal and others which are external to the many-valued logic under consideration, as we embed its original language into a more expressive metalanguage to deal with the satisfiability problem. In contrast to previous approaches to the same problem, our target language is fully classical, what turns the design of the resolution-based rules for a specific many-valued logic into a straightforward task. Correctness results, which are proved in detail in the present study, follow easily from results on classical resolution. Implementation of reasoning tools can be achieved by direct translation into classical propositional logic and making use of reliable existing automated provers. We illustrate the application of the method with examples.

Keywords: Multiple-valued Logics, Bivalent Semantics, Resolution Method

1 Introduction

Many-valued logics have just celebrated their centennial jubilee, and the computational proof method known as resolution is now commemorating its semicentennial birthday. We refrain from starting by an extensive overview of the contact points between the two areas, already covered in depth by other authors (check [11,2] and references therein). Instead, we proceed to clarify here the fundamental features of our current investigation by briefly commenting on the plan of the present paper.

Section 2 introduces logics in abstract and from a semantic viewpoint, explains that they can all be characterised by bivalent semantics, and then focuses on the class of finite-valued truth-functional logics. Emphasis is put on the standard (two-sided) notion of logical consequence, rather than on standard approaches to many-valued logics that are based on mere combinatorial manipulation of finite-valued algebras. Section 3 explains the algorithm that allows one to describe finite-valued logics in terms of statements written in a fully classical metalanguage in which only two signs are employed. Providing a proof-theoretical perspective on this requires a

Selected, revised papers will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
generalisation of the way that syntactical complexity is measured, to allow for analytic calculi to be extracted from such a description. The next subsequent section is dedicated to setting up a generic resolution-based proof method for our logics. Subsection 4.1 shows how to transform the already mentioned bivalent descriptions into a clausal format that is more appropriate for doing resolution. The corresponding transformation adds new variable symbols that help encoding the structure of the original statements, and preserves their satisfiability. As output we obtain object-level expressions that better reflect the above mentioned generalised notion of complexity. Subsection 4.2 introduces the inference rules of a hyper-resolution proof method that applies to the clauses produced by the latter transformation. This method lies in between internal proof systems that take advantage of syntactic features of the original logics and external proof systems that formalise reasoning about the logics in a classical logical framework. We finish by some comments on what has been achieved and how the present investigation may be further extended.

2 Every Many-valued Logic is Bivalent

This section contextualises our present study, and explains what we mean by a ‘logic’, from both an abstract and a semantic perspective. Special focus is put on the truth-functional case. We also introduce here an appropriate classical meta-language for describing a collection of valuations, and in the following section we discuss how to use it in implementing a computationally useful account of the so-called Suszko’s Thesis (check [3] and references therein), according to which “every logic has but two logical values”.

Definition 2.1 [Syntax] Let a (propositional) signature \( \Sigma \) be the union of a family \( \{ \Sigma_m \}_{m \in \mathbb{N}} \) of constructors, where each set \( \Sigma_m \) contains only function symbols of arity \( m \), and where \( \Sigma_i \) and \( \Sigma_j \) are assumed disjoint whenever \( i \neq j \). Let \( \mathcal{A} \) be a denumerable collection of symbols called (atomic) variables, assumed to be disjoint from the signature \( \Sigma \). The nullary symbols in \( \Sigma_0 \) are sometimes called truth symbols. A (propositional) language \( S \) is recursively generated in the usual way, by considering its composite formulae to be of the form \( \odot(\varphi_0, \varphi_1, \ldots, \varphi_m) \), for some \( \odot \in \Sigma_{m+1} \). In the latter case, the symbol \( \odot \) is said to be the head connective and the formulae \( \varphi_k \), for \( 0 \leq k \leq m \), are dubbed the immediate subformulae of \( \odot(\varphi_0, \varphi_1, \ldots, \varphi_m) \). The formulae in \( \mathcal{A} \cup \Sigma_0 \) are said to be noncomposite, and have no proper subformulae. A canonical notion of formula complexity \( \text{cp} : S \rightarrow \mathbb{N} \) may then be defined by setting \( \text{cp}(\varphi) = 0 \) if \( \varphi \) is noncomposite, and \( \text{cp}(\varphi) = 1 + \max_{0 \leq k \leq m} \text{cp}(\varphi_k) \) if \( \varphi \) is composite and the \( \varphi_k \), for \( 0 \leq k \leq m \), are its immediate subformulae. A uniform substitution is an endomorphism on the set of formulae that maps each constructor into itself; it is uniquely defined as soon as the variables of the language are mapped into formulae. By \( \varphi[p \mapsto \psi] \) we will denote the result of uniformly substituting \( \psi \) for each occurrence of the variable \( p \) in the formula \( \varphi \). 

In the present study we will only consider languages generated by finite signatures.

Definition 2.2 [Logic] Fixed a propositional language \( S \), a logic \( \mathcal{L} \) is here defined as a structure \( \langle S, >_\mathcal{L} \rangle \), where the relation \( >_\mathcal{L} \subseteq 2^S \times S \) respects the following four abstract axioms, for arbitrary \( \Gamma \cup \Delta \cup \{ \varphi \} \subseteq S \):

136
Marcos and Nalon

\( (C1) \) if \( \varphi \in \Gamma \), then \( \Gamma \succ_{L} \varphi \)

\( (C2) \) if \( \Gamma \succ_{L} \varphi \), then \( \Gamma \cup \Delta \succ_{L} \varphi \)

\( (C3) \) if \( \Gamma \succ_{L} \delta \), for every \( \delta \in \Delta \), and \( \Delta \succ_{L} \varphi \), then \( \Gamma \succ_{L} \varphi \)

\( (C4) \) if \( \Gamma \succ_{L} \varphi \), then \( \varepsilon(\Gamma) \succ_{L} \varepsilon(\varphi) \), for any uniform substitution \( \varepsilon \)

Any \( \succ_{L} \) respecting the first three axioms is called a Tarskian consequence relation. We call substitution-invariance the property described by the last axiom. We say that the formulae \( \alpha \) and \( \beta \) are \( L \)-equivalent, and denote this by \( \alpha \approx_{L} \beta \), if both \( \alpha \succ_{L} \beta \) and \( \beta \succ_{L} \alpha \). The following replacement property may or may not be respected by a given logic, where \( \varphi \) denotes an arbitrary formula to be used as context, and \( p \) denotes an arbitrary variable taken as placeholder:

\( (C5) \) if \( \alpha \approx_{L} \beta \), then \( \varphi[p \mapsto \alpha] \approx_{L} \varphi[p \mapsto \beta] \)

Logics that respect the latter axiom are called congruential.

A very natural kind of denotational semantics for a logical language is one that employs certain distinctive collections of truth-values in defining the associated notions of entailment:

**Definition 2.3** [Semantics] A *valuation* for a language \( S \) is a mapping \( w : S \rightarrow V_{w} \), where \( V_{w} \) denotes a nonempty collection of truth-values containing a subset \( D_{w} \) of designated values. The values in \( V_{w} \backslash D_{w} \) are called undesignated. As usual, a valuation \( w \) is said to satisfy a formula \( \varphi \) if the former assigns a designated value to the latter; otherwise, we say that \( w \) falsifies \( \varphi \). We will call \( V(2) = \{F,T\} \) the set of logical values, for which we fix \( D(2) = \{T\} \). Any valuation over \( V_{2} \) is referred to as a bivaluation. The restriction \( w|_{A} \) of the valuation \( w \) to \( A \subseteq S \) is called an assignment of truth-values to the variables. A *semantics* for \( S \) is here simply a family \( \text{Sem} \) of valuations. A valid formula (a.k.a. tautology) of a given semantics is one that is not falsified by any valuation from this semantics; a formula is called unsatisfiable by a given semantics if all the corresponding valuations falsify it. Fixed a semantics \( \text{Sem} \) and given some \( \Delta \subseteq S \), denote by \( \text{Mod}(\Delta) \) the set of valuations of \( \text{Sem} \) that map all formulae of \( \Delta \) into designated values. A canonical entailment relation \( \models_{\text{Sem}} \subseteq 2^{S} \times S \) is then defined by setting \( \Gamma \models_{\text{Sem}} \varphi \) iff \( \text{Mod}(\Gamma) \subseteq \text{Mod}(\{\varphi\}) \), i.e., we say that \( \Gamma \) entails \( \varphi \) iff there is no valuation in \( \text{Sem} \) that simultaneously satisfies all formulae in \( \Gamma \) and falsifies the formula \( \varphi \).

It is easy to check that an entailment relation always respects axioms \( (C1), (C2) \) and \( (C3) \) of a Tarskian consequence relation.

Semantics may come in different flavors, differing on the way they happen to collect the appropriate valuations for a given language. It is interesting to observe that there is a precise sense in which all semantics of the kind entertained above may be said to have a ‘bivalent character’. Indeed, fix a semantics \( \text{Sem} \), and for each \( w \in \text{Sem} \) define the total mapping \( b_{w} : S \rightarrow V(2) \) such that \( b_{w}(\varphi) = T \) iff \( w(\varphi) \in D_{w} \). Let \( \text{Sem}(2) \) be the family \( \{b_{w}\}_{w \in \text{Sem}} \) of bivaluations. It is straightforward now to check that \( \Gamma \models_{\text{Sem}(2)} \varphi \) iff \( \Gamma \models_{\text{Sem}} \varphi \). This means that every ‘multiple-valued logic’ may also be given a characteristic bivalent semantics. Hereupon, we shall call \( \text{Sem}(2) \) the bivalent reduction of \( \text{Sem} \).

The following definition takes advantage of structural features of the language in defining the associated semantics:
Definition 2.4 [Truth-functionality] For some fixed signature $\Sigma$ and some fixed set $V$ of truth-values, a $\Sigma$-algebra over $V$ is a structure in which each $\odot \in \Sigma_m$, with $m \in \mathbb{N}$, is interpreted as an operation $\odot : V^m \rightarrow V$ with the same arity. The very language $S$ may be seen as a $\Sigma$-algebra, if we take $V = S$: this defines the so-called term algebra generated by the variables in $A$ over the propositional signature $\Sigma$. If a semantics $\text{Sem}$ is given by the set of all homomorphisms from a given term algebra into a fixed $\Sigma$-algebra $V$ with carrier $V$, mapping each constructor $\odot$ into the corresponding operation $\hat{\odot}$, such semantics is said to be truth-functional. An entailment relation $\models_{\text{Sem}}$ is immediately defined, as before, as soon as a certain set $D \subseteq V$ of designated values is fixed throughout the valuations constituting the semantics $\text{Sem}$.

Note that each valuation of a truth-functional semantics is uniquely defined by some assignment of truth-values to the atomic variables in $A$, and the value of a composite formula $\varphi$ under a valuation $w$ is defined by the values given by this same valuation to the immediate subformulae of $\varphi$, so that $w(\odot(\varphi_0, \varphi_1, \ldots, \varphi_m)) = \hat{\odot}(w(\varphi_0), w(\varphi_1), \ldots, w(\varphi_m))$. This is indeed the way in which a truth-functional semantics implements the so-called ‘Principle of Compositionality of Meaning’.

It is clear that a truth-functional semantics gives rise to a substitution-invariant consequence relation. We shall call a logic $\langle S, \models \rangle$ genuinely $\kappa$-valued if $\kappa$ is the cardinality of the smallest set of truth-values $V$ over which a truth-functional semantics $\text{Sem}$ may be based to the effect that $\models_{\text{Sem}}$ and $\models$ are coextensional. A finite-valued logic is any genuinely $\kappa$-valued logic for which the cardinal $\kappa$ is finite. It is clear that the bivalent reduction described above still applies in this particular context, and that any logic with a truth-functional semantics may be characterised thus by a collection of bivaluations. In case a logic $L$ is genuinely $\kappa$-valued, for some $\kappa > 2$, it is clear, though, that its bivalent reduction cannot be a truth-functional semantics.

To associate a convenient proof system to a logic described in semantic terms, a wise choice of metalanguage can make a great difference. One rather straightforward way of describing a truth-functional semantics proceeds by the use of labelled formulae, and a corresponding extension of the associated interpretation: given $\varphi \in S$, we say that a valuation $w : S \rightarrow V$ satisfies $X \models \varphi$ if $w(\varphi) = X$ (do bear in mind, though, that while the $X$ in $w(\varphi) = X$ denotes a value from $V$, the same symbol plays the role of a purely syntactical sign in $X \models \varphi$). This is the approach taken by the so-called ‘singletons-as-signs’ labelling discipline; according to an alternative discipline called ‘sets-as-signs’, $w$ is said to satisfy $X \models \varphi$ if $w(\varphi) \subseteq X$, where $X \subseteq V$. In such metalanguage we will also introduce a meta-conjunction represented by $\&$, a meta-disjunction represented by $\mid\mid$, a meta-implication represented by $\Longrightarrow$, all with the expected Boolean interpretations, and statements such as $(X_A \models \varphi_a \& X_B \models \varphi_b) \Longrightarrow (X_C \models \varphi_c \mid\mid X_D \models \varphi_d)$ will then have their obvious reading as axioms imposing restrictions on the class of valuations in $\text{Sem}$, namely: if a valuation $w \in \text{Sem}$ happens to satisfy both $X_A \models \varphi_a$ and $X_B \models \varphi_b$, then this valuation should satisfy $X_C \models \varphi_c$ or $X_D \models \varphi_d$. The meta-conjunctions and the meta-disjunctions are generalised in the standard way to give support to any finite number of arguments.

We introduce in the metalanguage the symbols $\odot$ and $\odot$, respectively, for meta-verum (standing for an arbitrary labelled tautology, or a 0-ary conjunction) and
meta-falsum (standing for an arbitrary labelled antilogy, or a 0-ary disjunction), and write a statement such as $X : \varphi \xrightarrow{\oplus} \emptyset$ to say that $X : \varphi$ is unsatisfiable, and a statement such as $\oplus \xrightarrow{\emptyset} X : \varphi$ to say that $X : \varphi$ is unfalsifiable. Metalinguistic statements in which all labels are restricted to one of only two possible signs are said to be bivalent. As we will see, any finite-valued logic may be semantically characterised in terms of bivalent statements. We will assume $\mathcal{V}(2) = \{ F, T \}$ to be the set of labels used in case we are talking about bivalent statements, and shall write $X^c$ to denote the conjugate of $X$, defined by setting $F^c = T$ and $T^c = F$.

A natural classical meta-negation may also be introduced as we are working with bivalent statements: we will write $\overline{D}$ to say that statement $D$ fails to be the case.

In particular, on what concerns the interaction between the meta-negation and the labelled formulae, we will assume that $\overline{X : \varphi} = X^c : \varphi$. For a bivalent semantics, whose valuations are all total functions on $\{ F, T \}$, it should be clear that the following statements are always satisfied: $\oplus \xrightarrow{(X : \varphi \mid X^c : \varphi)}$ and $(X : \varphi \& X^c : \varphi) \xrightarrow{\emptyset}$.

**Example 2.5** Let $\Sigma_0^A = \{ \perp \}$ and $\Sigma_0^B = \{ \Rightarrow \}$. Let $\mathcal{V}(Q)$ be the set of all rational numbers in the real-valued interval $[0, 1]$, and let $\overline{D} = \{ 1 \}$. Assume $\widehat{\land} = 0$ and $\widehat{\lor}(x, y) = \min(1, 1 - x + y)$. As a side effect of such truth-functional interpretations, one might venture describing the implication $\Rightarrow$ by various metalinguistic statements such as $0 : \varphi \mid 1 : \psi \xrightarrow{1} (\varphi \Rightarrow \psi)$ or $\frac{1}{2} : (\varphi \Rightarrow \psi)$. Note however that a labelled composite statement such as $\oplus \xrightarrow{1} (\varphi \Rightarrow (\varphi \Rightarrow \psi)) \Rightarrow (\varphi \Rightarrow \psi)$ is satisfied iff $w(\varphi) = 0$ or $w(\varphi) = 1$. Now, the original signature may be extended by considering $\Sigma_1^B = \{ \text{id}, \neg, \theta_a, \theta_b \}$ and $\Sigma_2^B = \{ \land, \lor \}$ and taking these new symbols to be abbreviations of formulae written in the original signature, namely: $\neg \varphi \triangleq \varphi \Rightarrow \perp$; $\text{id}(\varphi) \triangleq 0$; $\theta_a(\varphi) \triangleq 0$; $\theta_b(\varphi) \triangleq 1$; $\varphi \lor \psi \triangleq (\varphi \Rightarrow \psi) \Rightarrow (\psi \Rightarrow \varphi)$; $\varphi \land \psi \triangleq \neg(\neg\varphi \& \neg\psi)$. As illustrations, notice, that while the metalinguistic statement $X : (\varphi \land \psi) \& 1 : \varphi \xrightarrow{\emptyset} X : \psi$ is satisfied for every $X \in Q \cap [0, 1]$ the statement $(X : \theta_a(\varphi) \mid Y : \theta_b(\varphi)) \xrightarrow{\emptyset} (X : \varphi \& Y : \neg\varphi)$ is satisfied only for $X, Y \in \{ 0, 1 \}$. One way of enforcing the validity of the latter statement is by replacing $\mathcal{V}(Q)$ by $\{ 0, 1 \}$ (restricting thus the set of undesignated values to the singleton $\{ 0 \}$). Lukasiewicz's logics $L_n$, for $n \geq 2$, are obtained if we replace $\mathcal{V}(Q)$ by $\mathcal{V}(n) = \left\{ \frac{m}{n-1} : 0 \leq m \leq n - 1 \right\}$. Equivalently, to the same effect one could impose on the semantics the bivalent axiom $\oplus \xrightarrow{1} \varphi \mid 1 : \neg\varphi$. In the above hierarchy of logics, Classical Logic (CL) corresponds to $L_2$ — thus, it not only has a bivalent semantics (one may set $b_w(\varphi) = T$ if $w(\varphi) = 1$, and $b_w(\varphi) = F$ otherwise) but is indeed a genuinely 2-valued logic. □

It is interesting to notice how, in the case of $L_2$, every bivalent statement may be rewritten in a useful way with the help of the negation connective: to that effect one just has to substitute ‘1:–’ for every ‘0:–’ that appears as prefix of a labelled formula, and confirm by induction that the resulting metalinguistic expression is satisfied iff the original metalinguistic expression is satisfied — so, in particular, $(0 : \varphi \mid 1 : \psi) \xrightarrow{\emptyset} 1 : (\varphi \Rightarrow \psi)$ becomes $(1 : \neg\varphi \mid 1 : \psi) \xrightarrow{1} (\varphi \Rightarrow \psi)$. This could give support to a natural argument for claiming that the addition of signs containing ‘semantic information’ to formulae is ‘superfluous’ (in the above example, one might consider just omitting the prefix ‘1:’ that now appears in front of every object language expression), in view of the expressivity that the classical object language displays in internalising the classical metalinguistic information given by the labels.
In general, of course, this argument runs unaltered only for CL. At any rate, there is nothing really special about negation in the preceding argument: the exact same impression of superfluity would in fact be caused by invoking the connective \( \theta \), whose interpretation only happens to coincide with that of negation over \( V(2) \).

It is not too hard to provide a combinatorial argument to show that Lukasiewicz’s logics are so related that \(|L_m| \subseteq |L_n| \) iff \( n - 1 \) divides \( m - 1 \). From the above example, however, it might be hard to tell the exact difference, say, between the 2-valued implication and the 5-valued implication just by looking at the statements that describe them, not least because of the use of different collections of labels to describe the semantics of CL and the semantics of \( L_5 \). A clever and generic way of solving this particular difficulty, in fact, would be by describing \( L_5 \) in a way that better mimics the 2-valued semantics of CL. For that, of course, the bivalent reduction of the 5-valued logic will do. The real problem, in that case, would then be how best to use our metalanguage to describe the corresponding non-truth-functional semantics in a computationally useful way.

There is a well-studied algorithmic approach ([4,5]) to produce a description of the bivalent reduction of any finite-valued logic, and to using this description to provide uniform tableau-theoretic characterisations of such a logic, with associated proof strategies that ensure, in each case, termination of the proof-search tasks, and do it by either producing a proof or a counter-example to any given conjecture. This consists in a mechanised procedure in four steps that receives as input the full specification (syntax and semantics) of a finite-valued logic \( L \) and:

(A1) check if the object language of \( L \) is sufficiently expressive for the task;
(A2) produce a sufficiently expressive conservative extension of \( L \), if necessary;
(A3) axiomatise the semantics of \( L \) using bivalent statements expressed in the classical metalanguage under a particular (disjunctive) normal form;
(A4) use the bivalent statements to produce a sound and complete (tableau) proof system for \( L \).

In the present paper we will have our logics presented already in a language that is expressive enough for our purposes, thus bypassing steps (A1) and (A2), where we have no contribution here to make. Next, we will import the original step (A3), and then introduce a transformation function that helps in massaging the bivalent statements into a ‘clause form’ that is more appropriate to the use by the resolution method. Finally, we will redesign step (A4) in order to produce sound and complete uniform classic-like resolution calculi for all finite-valued logics.

### 3 The Bivalent Reduction

We briefly describe, in what follows, the bivalent statements produced by the above mentioned algorithm that outputs the bivalent reduction of any finite-valued logic. The key to the procedure is to find, first, a way of using the language of the given logic to distinguish between each pair of designated truth-values, and similarly to distinguish between each pair of undesignated truth-values. In that way, each value of the original truth-functional semantics will turn out to have a unique ‘binary print’ that sets it apart from all the other values. Such binary print will be expressed in terms of a suitable combination of logical values.
For the following definitions we fix a (sufficiently expressive) genuinely \( \kappa \)-valued logic \( \mathcal{L} \), for finite \( \kappa \), whose semantics \( \text{Sem} \) has \( V \) and \( D \), respectively, as its sets of truth-values and designated values. We recall that \( b_w \) denotes the bivaluation induced by the valuation \( w \in \text{Sem} \).

**Definition 3.1** [Binary print] A separating sequence \( \overrightarrow{\theta} = \langle \theta_0, \theta_1, \ldots, \theta_s \rangle \) for \( \mathcal{L} \) is a sequence of unary connectives (to be referred to as separators) with the property that for every pair of distinct valuations \( w_1, w_2 \in \text{Sem} \), that is, every pair of valuations such that \( w_1(p) \neq w_2(p) \) for some variable \( p \), there is some \( \theta_r \), for \( 0 \leq r \leq s \), such that \( b_{w_1}(\theta_r(p)) \neq b_{w_2}(\theta_r(p)) \). Given some \( x \in V \) assigned by some valuation \( w \) to a given formula \( \varphi \), by the binary print \( \overrightarrow{\theta}(x) \) we mean the unique sequence \( \overrightarrow{X} = \langle b_w(\theta_0(\varphi)), b_w(\theta_1(\varphi)), \ldots, b_w(\theta_s(\varphi)) \rangle \in \{ F, T \}^{s+1} \). If an \((s + 1)\)-long sequence of \( F \)'s and \( T \)'s does not correspond to any binary print of a truth-value in \( V \), we say that such sequence represents an absurd. \( \Box \)

Intuitively, an absurd corresponds to a semantically unobtainable scenario: an alleged logical description of an ‘algebraic value’ that is not to be found among the available truth-values of the \( \Sigma \)-algebra interpreting the language.

The following should be read having Ex. 2.5 on the background.

**Example 3.2** Very short separation sequences are promptly available for Classical Logic. There are indeed in this case no pair of (un)designated values to distinguish. Thus, one could consider 1-long sequences made of any connective \( \odot \in \Sigma^A_1 \cup \Sigma^B_1 \), in view of the fact that \( b_{w_1}(\odot(p)) \neq b_{w_2}(\odot(p)) \) whenever \( w_1(p) = 0 \) and \( w_2(p) = 1 \). Note, now, that \( \langle \text{id} \rangle \) is by itself not a separating sequence for \( \mathcal{L}_3 \), for it does not tell the two undesignated values apart. Any 2-long sequence of distinct unary connectives from our extended signature would however do the job equally well for \( \mathcal{L}_3 \). Separating sequences for other \( \mathcal{L}_n \)'s are forcibly longer. For \( \mathcal{L}_5 \), for instance, one could make do with the 4-long sequence \( \langle \text{id}, \neg, \theta_a, \theta_b \rangle \). \( \Box \)

To simplify matters, we will choose henceforth to set \( \theta_0(p) = p \). Strictly speaking, in this case \( \theta_0 \) will not really denote a unary connective, but it will be just as good, as its interpretation coincides in fact with the interpretation of the ‘identity connective’ \( \text{id} \). Obviously, \( b_{w_1}(\theta_0(p)) \neq b_{w_2}(\theta_0(p)) \) whenever \( w_1(p) \) denotes a designated value, and \( w_2(p) \) denotes an undesignated value. Thus, the other \( \theta_r \)'s (for \( r > 0 \)) in the separating sequence have the role of allowing us to distinguish between two truth-values that are both designated, as well as between two truth-values that are both undesignated. The very existence of a separating sequence means that congruentiality (axiom (C5) in Def. 2.2) always fail for sufficiently expressive truth-functional logics. Indeed, one can in general only be assured that two equivalent formulae may be replaced one by another salva veritate when every valuation assigns to them exactly the same truth-value, for otherwise there will be some context that distinguishes these equivalent formulae. However, the replacement property is obviously fully enjoyed by our classical metalanguage, and we will in the future be using it often, at the meta-level, in replacing a labelled formula by any classically equivalent labelled formula.

The above definitions lead us now very naturally to an extension of the notion of formula complexity:
Definition 3.3 [Generalized measure of complexity] Formulae of the form $\theta_r(\varphi)$, where $\theta_r$ is a separator and $\varphi$ is noncomposite, are called basic. A generalised notion of formula complexity $gcp : S \rightarrow \mathbb{N}$ is defined by setting $gcp(\varphi) = 0$ if $\varphi$ is basic, and $gcp(\varphi) = 1 + \max_{0 \leq k \leq m} gcp(\varphi_k)$, if $\varphi$ has the form $\theta_r(\bigcirc(\varphi_0, \varphi_1, \ldots, \varphi_m))$, for $\bigcirc \in \Sigma_{m+1}$ and $0 \leq r \leq s$. Formulae with positive complexity $gcp$ are said to be analysable. The complexity measure may be extended to labelled formulae by setting $gcp(X; \varphi) = gcp(\varphi)$.

Note that, in the present approach, neither labels nor separators contribute to an increase in formula complexity, and in general we have that $gcp(\varphi) \leq cp(\varphi)$.

Let $\overrightarrow{b}(X) = \langle X_0, X_1, \ldots, X_s \rangle$ be the binary print of some truth-value $X \in \mathcal{V}$. Note that there is a sense in which the generic metalinguistic statement $X: \varphi$ may be assumed to be described by the bivalent statement $\&_{r=0}^s X_r: \theta_r(\varphi)$. Indeed, the former statement is satisfied iff the latter is satisfied, modulo the respective labellings. In other words: $w(\varphi) = X$ iff $b_u(\theta_r(\varphi)) = X_r$ for every $0 \leq r \leq s$. We shall refer to $\&_{r=0}^s X_r: \theta_r(\varphi)$ as $V(\varphi, X)$. Considering next an analysable formula $\varphi$ of the form $\theta_r(\bigcirc(\varphi_0, \varphi_1, \ldots, \varphi_m))$ and a logical value $X$, we shall call $F^\theta_{X, \bigcirc}$ the set of all tuples of values from $\mathcal{V}$ that the subformulae $\varphi_0, \varphi_1, \ldots, \varphi_m$ may be assigned by a valuation $w \in \text{Sem}$ in order to guarantee that $b_u(\varphi) = X$.

Aided by the separators, we now set ourselves the goal of describing the behaviour of analysable formulae in terms of formulae with lower complexity measure. In particular, to each label $X \in \{F, T\}$, each separator $\theta_r \in \Sigma^\theta$ and each $m$-ary nonseparator connective $\bigcirc$ from the signature of $\mathcal{L}$ we will associate the following $B$-statement:

$$X: \theta_r(\bigcirc(p_1, p_2, \ldots, p_m)) \implies \|_{(x_1, x_2, \ldots, x_m) \in R^\theta_{X, \bigcirc}} (\&_{k=1}^m V(p_k, \overrightarrow{b}(x_k))) \qquad (F^\theta_{X, \bigcirc})$$

In addition to the above, we should note that all the $(s + 1)$-long binary sequences of $F$’s and $T$’s that do not correspond to binary prints of truth-values in $\mathcal{V}$ in fact describe impossible semantic scenarios. As it happens, however, it is often the case that partial knowledge about a given binary sequence gives enough information for us to conclude that it represents an absurd. In other words, maybe we do not know all elements of a given binary sequence at a certain stage of development of our reasoning, but we know enough to be able to conclude that any way of completing this sequence leads to an absurd. Accordingly, it is useful to entertain certain ‘minimal unobtainable partial binary sequences’. The idea is to add a symbol for an ‘undefined’ logical value $\uparrow$ and call some sequence $\overrightarrow{y} \in \{F, T\}^{s+1}$ unobtainable already when any $(s + 1)$-long fully defined sequence of $F$’s and $T$’s that coincides with the former sequence in all positions that do not contain undefined symbols represents an absurd, that is, when no extension of $\overrightarrow{y}$ represents an algebraic value from the original semantics. As expected, a minimal unobtainable such sequence is one that does not properly extend another unobtainable sequence.

Example 3.4 To illustrate the issue about partial binary sequences, note that if $L_5$ is considered over the separating sequence $\langle \text{id}, \neg, \theta_a, \theta_b \rangle$, as in Ex. 3.2, its truth-values will be individualised by 5 out of 16 possible tuples of $F$’s and $T$’s. One (minimal) description of the 11 unobtainable tuples might be given by the following partial binary sequences: $\langle \uparrow, \uparrow, F, F \rangle$, $\langle \uparrow, T, \uparrow, F \rangle$, $\langle \uparrow, T, T, \uparrow \rangle$, $\langle T, \uparrow, F, \uparrow \rangle$, and $\langle T, \uparrow, \uparrow, T \rangle$.
Given a minimal unobtainable sequence \( Y \in \{ F, T \}^{s+1} \), by \( \text{dom}(Y) \) we denote the set \( \{ 0 \leq r \leq s : Y_r \neq \top \} \). To any such sequence \( Y \) we will associate the following \( U \)-statement:
\[
& \forall r \in \text{dom}(Y) \; Y_r; \theta_r(p_0) \implies \top \\
\text{(U)}
\]
We may say that a minimal unobtainable binary sequence \( Y \) covers any \((s+1)\)-long binary sequence \( X \) such that \( \text{dom}(Y) \subseteq \text{dom}(X) \).

The above \( B \)-statements and \( U \)-statements allow us to describe a very convenient list of bivalent statements to use in characterising a given finite-valued logic.

**Definition 3.5** [Bivalent statements induced by \( L \)] Let \( L \) be a finite-valued logic over a signature \( \Sigma \) and a set of truth-values \( \mathcal{V} \), let \( \theta \) be a separating sequence for \( L \), call \( \Sigma^\theta \subseteq \Sigma_1 \) the set of separators in \( \theta \), and let \( \{ Y_j \}_{j \in \lambda} \), for some \( \lambda \in \mathbb{N} \), be a family of minimal unobtainable binary sequences that jointly cover all the binary prints that do not correspond to truth-values in \( \mathcal{V} \). The bivalent semantics that we shall call \( B(L, \theta) \) is described by the collection of all \( B \)-statements \( B^\theta_X \), for each label \( X \in \{ F, T \} \), for each \( \theta, \epsilon \in \Sigma^\theta \) and each \( \top, \epsilon \in \Sigma \setminus \Sigma^\theta \), together with the collection of all \( U_{Y_j} \)-statements, for \( j \in \lambda \).

Note that \( B^\theta_F \cup B^\theta_T = \mathcal{V}^{m+1} \), for \( \top, \epsilon \in \Sigma_m \) (and also that \( B^\theta_F \cap B^\theta_T = \emptyset \), but it may occur that one of these two sets is empty. In that case, we will have an empty disjunction on the right-hand side of a \( B \)-statement \( B^\theta_X \), and this obviously amounts to a ‘degenerate’ \( B \)-statement that looks more like a \( U \)-statement, given that it describes a semantically unobtainable scenario.

\[
\begin{array}{|c|c|}
\hline
(U(T.F)) & (T;p_0 \land F;\theta(p_0)) \implies \top \\
(B^\theta_T) & T;\top \implies \top \\
(B^\theta_F) & T;\bot \implies \top \\
(B^\theta_0) & F;\theta(\theta(p_0)) \implies F;\theta(p_0) \\
(B^\theta_1) & T;\theta(\theta(p_0)) \implies T;\theta(p_0) \\
(B^\theta_2) & F;p_0 \lor p_1 \implies (T;p_0 \land F;p_1) \lor (T;\theta(p_0)) \lor (T;p_1) \\
(B^\theta_3) & T;p_0 \lor p_1 \implies (F;p_0 \land T;\theta(p_1)) \lor (F;p_1) \lor T;p_1 \\
(B^\theta_4) & F;\theta(p_0 \lor p_1) \implies (T;p_0 \land F;\theta(p_1)) \\
(B^\theta_5) & T;\theta(p_0 \lor p_1) \implies F;p_0 \lor T;\theta(p_1) \\
\hline
\end{array}
\]

Fig. 1.

The following are among the main results of [5]:

**Theorem 3.6** (Soundness & Completeness) The bivalent semantics described by \( B(L, \theta) \) contains the same bivaluations included in the bivalent reduction \( \text{Sem}(2) \) of \( L \).

**Theorem 3.7** (Effectiveness) Let \( b \in \text{Sem}(2) \), and let \( \varphi \) be an analysable formula. Let \( A(\varphi) \) be the set of variables that occur in \( \varphi \). Then the value of \( b(\varphi) \) is uniquely determined from the values of \( b(\theta_r(p)) \) for \( 0 \leq r \leq s \) and \( p \in A(\varphi) \). Moreover, the value of \( b(\varphi) \) may be effectively computed using the \( B(L, \theta) \) statements.

The perspicuous reader will surely have suspected that this provides a way of extracting from \( B(L, \theta) \) the rules characterising an analytic tableau system. In the
next section we will show how such description can be transformed into a description that is more appropriate for working with resolution calculi.

The following illustration builds on previous examples, and will be further explored in the next section.

**Example 3.8** Fig. 1 contains a set of bivalent axioms for $L_3$ produced by the above described algorithm, using $\langle \theta_0, \theta_a \rangle$ as a separating sequence. To simplify notation, we omit the subscript in $\theta_a$.

## 4 Resolution Calculus

This section introduces a proof method for finite-valued logics through backward reasoning in the form of a clausal resolution-based proof method, a refutational procedure applied to formulae into a specific Conjunctive Normal Form. Clausal resolution usually takes a suitable normal form for the classical negation of a formula to be tested for satisfiability and then a (set of) rules based on the Resolution Principle are applied until the empty clause is found or no new clauses can be generated. If the empty clause is found, the original formula is unsatisfiable; otherwise, the formula is satisfiable and we can build a model that witnesses its satisfiability. We also take here the clausal approach, but instead of working from the original semantics of a given finite many-valued logic, we use the bivalent semantics for such a logic, described in Section 3, to produce the normal form. This way, the calculus we shall present here works over a set of clauses in the metalanguage representing the bivalent semantics. The calculus is parametrised by the language and the separating sequence used to produce the metalinguistic statements for a given logic. We will denote by $\text{RES}_B(L, \vec{\theta})$ our resolution calculus for the bivalent statements $B(L, \vec{\theta})$ that describe the many-valued logic $L$ with the separating sequence $\vec{\theta}$.

### 4.1 Normal Form

Let $L = \langle S, >_L \rangle$, where $S$ is recursively defined over a finite signature $\Sigma = \bigcup_{k \in \mathbb{N}} \Sigma_k$, be a fixed finite-valued logic with a truth-functional semantics $\text{Sem}$. Let $B(L, \vec{\theta})$ be the description of the bivalent semantics $\text{Sem}(2)$ corresponding to our bivalent reduction of $\text{Sem}$.

From Def. 3.5 we note that the bivalent statements describing a given semantics have their right-hand side in Disjunctive Normal Form. However, clausal resolution works over conjunctions of clauses (i.e., meta-disjunctions of labelled basic formulae) representing the Conjunctive Normal Form of a formula. More precisely, the normal form that is obtained from our bivalent statements, henceforth to be called $\text{CNF}_{BS}$, consists in metalinguistic statements of the form $\land_{i=0}^{a} (\big|_{j=0}^{b} X_{ij} \cdot \varphi_{ij})$ where $a, b_i \in \mathbb{N}$, $X_{ij} \in \{T, F \}$ and each $\varphi_{ij}$ is a basic formula. We note from Def. 3.3 that the disjuncts $X_{ij} \cdot \varphi_{ij}$ cannot be further analysed, and play thus the role of ‘literals’ in classical resolution. Furthermore, as the meta-conjunction is associative, commutative and idempotent, we may treat a formula into $\text{CNF}_{BS}$ more simply as a set of clauses.

There are alternative ways of producing the conjunctive normal form starting from the $B$-statements $B_X^{\ominus}$. One of these is to apply the usual distribution rules to such a metalinguistic formula, but that might give rise to an exponential increase in
the size of the formula. Another way of transforming a formula into CNF$_{BS}$ is to apply the meta-negation to formulae in $B^\theta_Xe^\Theta$ followed by applications of De Morgan at the level of the metalanguage. However, it has been shown that ‘language-preserving transformations’ like these might lead to larger sets of clauses (cf. Ex. 4.6 in [2]) than the so-called ‘structure-preserving transformations’. In contrast, the latter kind of transformations introduce abbreviations, in the form of new atomic variables and bi-implications, in order to help producing a normal form. This is the main approach chosen, for instance, in [6], and also entertained in [2]. In the present study, we combine both kinds of transformation, but taking advantage of the structure of B-statements, where the right-hand side of meta-implications are already meta-disjunctions, and we use renaming to replace analysable formulae appearing as disjuncts. Applying at the metalinguistic level the results from [8], if a formula $\varphi$ is a subformula of $\psi$, that is, if $\psi$ has the form $\psi[p \rightarrow \varphi]$, then the labelled formula $X:\psi' \& X:((t_\varphi \implies \varphi) \& X:((\varphi \implies t_\varphi))$, where $\psi' = \psi[p \rightarrow t_\varphi]$ for some fresh atomic variable $t_\varphi$, can be used to replace the formula $X:\psi$ whilst preserving satisfiability.

The ‘meta-bi-implication’ $X:(t_\varphi \implies \varphi) \& X:((\varphi \implies t_\varphi)$ is often referred to as the definition of $\varphi$. In the present study, as in [6], we restrict renaming to formulae of positive polarity, that is, formulae that occur in the scope of an even number of meta-negations. Thus, only one side of the definition of $\varphi$ is needed, namely the meta-implication $X:(t_\varphi \implies \varphi)$. Again, satisfiability is preserved and the resulting normal form is shorter (cf. [8]). Differently from [6] and [2], the transformation into CNF$_{BS}$ does not produce many-valued labelled formulae (either following the efficient labelling discipline ‘sets-as-signs’, or the straightforward ‘singletons-as-signs’), but directly produces classic-like two-signed formulae. To some extent, our approach is closer to [7], where the satisfiability problem for many-valued logics is reduced to the satisfiability problem in classical propositional logic, but we restrict ourselves to the two labels that represent the underlying logical values. As a pleasant consequence, existing satisfiability procedures may be used to test a set of clauses without further ado. We note, however, that here we also need to take into consideration the U-statements, which provide the restrictions under which a meta-conjunction of labelled-formulae is meaningful. The U-statements will originate appropriate resolution-based rules, in Section 4.2. In what follows we define the necessary prior transformation into CNF$_{BS}$.

Our transformation function $\tau$ takes a labelled formula $X:\varphi$ as input and converts it into a normal form through recursive applications of rewriting and renaming. To each B-statement of the form $X_0:\varphi_0 \implies (((\prod_{i=1}^n X_i:\varphi_i))$ we associate the following rewrite rule: $\tau(X_0:\varphi_0) \iff \tau(((\prod_{i=1}^n X_i:\varphi_i))$. If $\odot$ is the head connective in $\varphi_0$, we indicate the corresponding rewrite rule as $\tau_{X_0}^\odot$. The transformation distributes over meta-conjunctions and over meta-disjunctions, as follows:

\[ \tau((\prod_{i=1}^n X_i:\varphi_i)) \iff \&_{i=1}^n \tau(X_i:\varphi_i). \quad (\tau^\&) \]

\[ \tau(((\prod_{i=1}^n X_i:\varphi_i)) \iff \prod_{i=1}^n \tau(X_i:\varphi_i). \quad (\tau^\prod) \]

The next rule renames meta-conjunctions that appear within meta-disjunctions:

\[ \tau(\psi \mid X:\varphi) \iff \tau(\psi \mid T:t_\varphi) \& \tau(T:t_\varphi \implies X:\varphi) \quad (\tau^\text{ren}) \]

where $\psi$ is a labelled formula, $X:\varphi$ is a meta-conjunction and $t_\varphi$ is a fresh atomic variable. Note that meta-disjunctions are associative and commutative. Thus, $\tau^\text{ren}$ applies to any disjunct that is not a labelled basic formula. Meta-conjunctions on
1. \(T:t_1 \or T:t_9\)
2. \(F:t_1 \or T:t_2 \or T:t_8 \or T:p\)
3. \(F:t_1 \or F:p\)
4. \(F:t_2 \or T:t_3 \or T:t_6\)
5. \(F:t_2 \or \theta(p)\)

\[\text{Logic } L \text{ on a language } S, \] and let \(t_{Uj\lambda p}\), for some \(\lambda \in \mathbb{N}\), be the family of \(U\)-statements in this description.

The right-hand side of meta-implications are rewritten in the usual way:
\[\tau(T:t \implies \&_{i=1}^n X_i:\varphi_i) \implies \&_{i=1}^n \tau(T:t \implies \tau(X_i:\varphi_i)) \quad (\tau \implies \&)\]

The final rewrite rule transforms meta-implications into meta-disjunctions:
\[\tau(T:t \implies D) \longmapsto \begin{cases} 
F:t \implies D, & \text{if } D \text{ is a clause} \\
\tau(T:p \implies \tau(D)), & \text{otherwise} 
\end{cases} \quad (\tau \implies \lor)\]

As the base case for the transformation function, we set:
\[\tau(X:\varphi) \implies X:\varphi, \text{ if } \varphi \text{ is a basic formula.} \quad (\tau^b)\]

Clauses are kept in simplified form, i.e. the following simplification rules apply at all steps of the transformation (where \(D\) is a clause and \(\varphi\) is a basic formula):
\[
\begin{align*}
\sigma(D \parallel X:\varphi \parallel X:\varphi) & \implies \sigma(D \parallel X:\varphi) \\
\sigma(D \parallel X:\varphi \parallel X^c:\varphi) & \implies \bot \\
\sigma(D \parallel \bot) & \implies \sigma(D) \\
\sigma(D \parallel \bot) & \implies \sigma(D)
\end{align*}
\]

The following lemma shows that the transformation into \(\text{CNF}_{BS}\) is correct.

**Lemma 4.1** Let \(\mathcal{L}\) be a finite-valued logic and \(S\) be its language. Let \(B(\mathcal{L}, \overline{\theta})\) be the set of bivalent statements associated with \(\mathcal{L}\) and the separating sequence \(\overline{\theta}\), and let \(\varphi\) be a formula in \(S\). Then, \(\varphi\) is satisfiable if, and only if, \(\tau(T:\varphi)\) is satisfiable.

**Proof.** The rewrite rules given by \(\tau\) are based on the equivalences given in [5] (case of \(\overline{\theta}_0\)), on classical equivalences for all other rewrite rules (\(\tau^\&\), \(\tau^\lor\), \(\tau \implies \&\), and \(\tau \implies \lor\)), and on classical renaming for the transformation rule \(\tau^{ren}\). As both kinds of transformation, namely replacement and renaming, preserve satisfiability (cf. [8]), we conclude that \(\varphi\) is satisfiable if and only if \(\tau(T:\varphi)\) is also satisfiable.

**Example 4.2** The formula \(\varphi_0 = ((p \lor (p \lor \bot)) \lor p) \lor p\) is valid in \(L_3\). Figure 2 shows the set of clauses resulting from the transformation of \(F:\varphi_0\) into \(\text{CNF}_{BS}\), where the transformation function is based on the set of bivalent statements for \(L_3\) with separating sequence \(\langle \theta_0, \theta_0 \rangle\), given in Fig. 1. Tautologies have been suppressed. On the right-hand side, we present the definitions of the new atomic variables introduced along the translation.

\[\square\]

<table>
<thead>
<tr>
<th>New Variable</th>
<th>Renames</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t_1)</td>
<td>(T:\left((p \lor (p \lor \bot)) \lor p\right) \land F:p)</td>
</tr>
<tr>
<td>(t_2)</td>
<td>(F:\left((p \lor (p \lor \bot)) \land T:θ(p)\right))</td>
</tr>
<tr>
<td>(t_3)</td>
<td>(T:p \land F:(p \lor \bot))</td>
</tr>
<tr>
<td>(t_4)</td>
<td>(T:p \land F:⊥)</td>
</tr>
<tr>
<td>(t_5)</td>
<td>(T:θ(p) \land F:θ(\bot))</td>
</tr>
<tr>
<td>(t_6)</td>
<td>(T:θ(p) \land F:θ(\bot))</td>
</tr>
<tr>
<td>(t_7)</td>
<td>(T:p \land F:θ(\bot))</td>
</tr>
<tr>
<td>(t_8)</td>
<td>(T:p \land F:θ(\bot))</td>
</tr>
<tr>
<td>(t_9)</td>
<td>(T:θ((p \lor (p \lor \bot)) \lor p) \land F:θ(p))</td>
</tr>
</tbody>
</table>

**4.2 Inference Rules**

Let \(B(\mathcal{L}, \overline{\theta})\) be the bivalent description of a finite-valued logic \(\mathcal{L}\) on a language \(S\), and let \(\{U_j\}_{j \in \lambda}\), for some \(\lambda \in \mathbb{N}\), be the family of \(U\)-statements in this description.
Let $X: \varphi$ be a labelled formula with $\varphi \in \mathcal{S}$. Let $\Phi$ be the set of clauses obtained by transforming $X: \varphi$ into the corresponding CNF$_{BS}$ based on $B(\mathcal{L}, \overline{\theta})$. The resolution calculus for $\mathcal{L}$ will comprise a binary resolution rule $(RES)$, which is a syntactical variation of the usual classical (binary) resolution rule [10], and a set of hyper-resolution inference rules, named $(RES_{U_j})$, for $j \in \lambda$, to deal with the valuation restrictions related to the corresponding $U_j$-statements. Hyper-resolution (cf. [9]) is a refinement of the resolution method, which combines several binary resolution steps, thus avoiding the generation of intermediate clauses (and the interaction between them) when searching for a proof. Recall that $U$-statements express the binary sequences which are unobtainable as binary prints of the original truth-values of $\mathcal{L}$. Hence, the meta-conjunction of labelled formulae on the left-hand side of the meta-implication in a $U$-statement corresponds to a logical absurd in the semantics of the metalanguage. The resolution rules are given in Fig. 3, where $D_i$ is a clause, $X_i \in \{T, F\}$, and $\varphi, \varphi_i$ are basic formulae (where $1 \leq i \leq n_j$, $j \in \lambda$, and $n_j$ is the number of meta-conjuncts in $U_j$). Premises of the resolution-based inference rules are called parent clauses. Conclusions in such rules are called resolvents. Also, we refer to $\{T: \varphi, F: \varphi\}$ (resp. $\{X_1: \varphi_1, \ldots, X_{n_j}: \varphi_{n_j}\}$) occurring in the parent clauses of $(RES)$ (resp. $(RES_{U_j})$) as a contradictory set of formulae. Labelled formulae in a contradictory set of formulae are said to be resolved by the respective inference rule.

\[
\begin{align*}
(RES) & \quad D_1 \parallel T: \varphi \\
(RES_{U_j}) & \quad D_1 \parallel X_1: \varphi_1 \\
& \quad D_2 \parallel F: \varphi \\
& \quad D_1 \parallel D_2 \\
& \quad \vdots \\
& \quad D_{n_j} \parallel X_{n_j}: \varphi_{n_j}
\end{align*}
\]

\[
\begin{align*}
& \quad D_1 \parallel \ldots \parallel D_{n_j} \\
& \quad \text{for each } U_j = X_1: \varphi_1 \& \ldots \& X_{n_j}: \varphi_{n_j} \Rightarrow \emptyset
\end{align*}
\]

Fig. 3. Resolution-based Rules for $\text{RES}_{B(\mathcal{L}, \overline{\theta})}$.

**Definition 4.3** [Derivations & Refutations] Let $\Phi$ be a set of clauses in CNF$_{BS}$. A derivation for $\Phi$ is a sequence $\Phi_0, \Phi_1, \ldots$ of sets of clauses in CNF$_{BS}$, with $\Phi_0 = \Phi$ and for all $i > 0$, $\Phi_{i+1} = \Phi_i \cup \{D\}$, where $D$ is a clause (in simplified form) obtained by an application of either $(RES)$ or $(RES_{U_j})$ in $\text{RES}_{B(\mathcal{L}, \overline{\theta})}$ to clauses in $\Phi_i$. Repeated formulae are not added to the next step in a derivation, i.e. we require that $(\Phi_{i+1}\setminus \Phi) \neq \emptyset$, and that $D$ is not a tautology. A refutation for a set of clauses $\Phi$ is a finite derivation for $\Phi$, $\Phi_0, \ldots, \Phi_m$, with $m \in \mathbb{N}$, such that $\emptyset \in \Phi_m$. □

Next, we show an example of the application of the method.

**Example 4.4** The formula $(((p \supset (p \supset \bot)) \supset p) \supset p)$ is valid in $L_3$. We present a refutation for the set of clauses resulting from the transformation of $F:(((p \supset (p \supset \bot)) \supset p) \supset p)$ into CNF$_{BS}$ given in Example 4.2. As shown in Table 1, there is only one $U$-statement for the bivalent semantics of $L_3$, namely, $(T: \varphi \& F: \theta(\varphi)) \Rightarrow \emptyset$. Thus, the corresponding hyper-resolution rule has the form: from $D \parallel T: \varphi$ and $D' \parallel F: \theta(\varphi)$ infer $D \parallel D'$, where $D$ and $D'$ are clauses and $\varphi$ is a formula of $L_3$. In the following, we only show the clauses that are relevant in the refutation. Justification is given on the right: Numbers correspond to the parent clauses and the application of the hyper-resolution rule is indicated by $U$ (otherwise, $(RES)$ is applied). We also indicate in each case the list of labelled formulae being resolved.

147
A derivation terminates iff either the empty clause is derived or no new clauses can be derived by further application of the resolution rules of $\text{RES}_{B(\mathcal{L}, \overline{\theta})}$. From Definition 4.3, recall that we do not add repeated clauses nor tautologies to the generated set of clauses. These restrictions, together with simplification, are usual in resolution-based proof methods and help to establish termination, as follows.

**Theorem 4.5 (Termination)** Let $\Phi$ be a set of clauses in CNF$_{BS}$. Then any derivation for $\Phi$ in $\text{RES}_{B(\mathcal{L}, \overline{\theta})}$ terminates.

**Proof.** Firstly, note that none of the inference rules introduces new labelled basic formulae in the clause set. As there are only a finite number of such formulae occurring in $\Phi$, only a finite number of CNF$_{BS}$ clauses can be built. Formally, if $n$ is the number of basic formulae occurring in $\Phi$, there are only $3^n$ possible different meta-disjunctions (modulo reordering) that can be built: either the labelled basic formula $X_i \varphi$ does not occur in the clause, or it occurs in the form $T_i \varphi$, or else it occurs in the form $F_i \varphi$ (as meta-disjunctions are in simplified form). Also, for any derivation $\Phi_0, \Phi_1, \ldots, \Phi$, by the definition of a derivation, we have that $\Phi_{i+1}$ must contain a new clause as compared to $\Phi_i$. Thus, there must be some $m \in \mathbb{N}$ such that either the empty clause is in $\Phi_m$, or $|\Phi_m| \leq 3^n$ and no new further clauses can be generated. Therefore, any derivation for $\Phi$ in $\text{RES}_{B(\mathcal{L}, \overline{\theta})}$ terminates.

The next result establishes soundness of the resolution method.

**Theorem 4.6 (Soundness)** The resolution calculus $\text{RES}_{B(\mathcal{L}, \overline{\theta})}$ for $B(\mathcal{L}, \overline{\theta})$, based on the rules (RES) and (RES$_{U_j}$), for each $U_j$-statement in $B(\mathcal{L}, \overline{\theta})$, is sound.

**Proof.** We start by establishing the soundness of each inference rule in $\text{RES}_{B(\mathcal{L}, \overline{\theta})}$. Soundness of (RES) follows from the results in [10]. For the soundness of (RES$_{U_j}$), assume all premises hold. Then, by definition of satisfiability, the meta-conjunction of those premises is also satisfiable. It is a straightforward exercise to show that $\&_{i=1}^{n_j}(D_i \mid X_i \varphi_i)$ is semantically equivalent to $(\&_{i=1}^{n_j} D_i) \Rightarrow (\&_{i=1}^{n_j} X_i \varphi_i)$, where $D_i$ is the meta-negation of $D_i$. From the results in [5], by the $U_j$-statement $\&_{i=1}^{n_j} X_i \varphi_i \Rightarrow \Theta$, we have that $\&_{i=1}^{n_j} X_i \varphi_i$ is unsatisfiable. Thus, by the semantics...
of the meta-implication, we conclude that \( \land_{i=1}^{n_j} \overline{D_i} \) is unsatisfiable, and by the semantics of the meta-conjunction and meta-negation, we obtain that \( \lor_{i=1}^{n_j} D_i \), the resolvent of \( (\text{RES}_{U_j}) \), is satisfiable. Now, let \( \Phi \) be a satisfiable set of clauses and let \( \Phi_0, \Phi_1, \ldots \) be a derivation for \( \Phi \). As all inference rules are sound, by an easy induction on the length of a derivation, every \( \Phi_i, i \geq 0 \), is also satisfiable. Thus, the resolution calculus based on (RES) and \( (\text{RES}_{U_j}) \) is sound.

As expected, the completeness result only depends on showing that the hyper-resolution rule can be simulated by binary resolution.

**Lemma 4.7** Let \( \mathcal{B}(\mathcal{L}, \overline{\vartheta}) \) be the bivalent semantics of \( \mathcal{L} \) with separating sequence \( \overline{\vartheta} \) and \( \text{RES}_{\mathcal{B}(\mathcal{L}, \overline{\vartheta})} \) the resolution calculus based on \( \mathcal{B}(\mathcal{L}, \overline{\vartheta}) \). Let \( (\text{RES}_{U_j}) \) be the hyper-resolution rule corresponding to the \( U_j \)-statement \( \land_{i=1}^{n_j} X_i \land \overline{\varphi_i} \Rightarrow \Theta \) in \( \mathcal{B}(\mathcal{L}, \overline{\vartheta}) \). Then, \( (\text{RES}_{U_j}) \) can be simulated by binary resolution.

**Proof.** The proof is straightforward. Assume \( \Phi \) is the set of clauses \( \{ (D_1 \mid X_1 \land \varphi_1), (D_2 \mid X_2 \land \varphi_2), \ldots, (D_{n_j} \mid X_{n_j} \land \varphi_{n_j}) \} \), i.e. the premises in \( (\text{RES}_{U_j}) \). Note that the meta-negation of \( \land_{i=1}^{n_j} X_i \land \varphi_i \) corresponds to the unfalsifiable meta-disjunction \( \lor_{i=1}^{n_j} X_i \land \overline{\varphi_i} \). Call the latter statement \( D \). Take \( \Phi' = \Phi \cup \{ D \} \). Construct a derivation \( \Phi'_0, \Phi'_1, \ldots \) for \( \Phi' \) by taking \( \Phi'_0 = \Phi' \), where \( D'_i \) is the resolvent of \( D \) and \( (D_1 \mid X_1 \land \varphi_1) \) by (RES). For the remainder of the construction, take \( \Phi'_{i+1} = \Phi'_i \cup \{ D'_{i+1} \} \), where \( D'_{i+1} \) is the resolvent of \( D'_i \) and \( (D_{i+1} \mid X_{i+1} \land \overline{\varphi_{i+1}}) \) by (RES). Thus, \( \Phi'_{n_j} \) contains \( \lor_{i=1}^{n_j} D_i \), the same clause as the resolvent of \( (\text{RES}_{U_j}) \) applied to \( \Phi \).

**Theorem 4.8 (Completeness)** Let \( \mathcal{B}(\mathcal{L}, \overline{\vartheta}) \) be the bivalent semantics of \( \mathcal{L} \) with separating sequence \( \overline{\vartheta} \). The resolution calculus, \( \text{RES}_{\mathcal{B}(\mathcal{L}, \overline{\vartheta})} \), based on (RES) and the family of rules \( (\text{RES}_{U_j}) \), for each \( U_j \)-statement in \( \mathcal{B}(\mathcal{L}, \overline{\vartheta}) \), is complete.

**Proof.** Immediate from Lemma 4.7 and the completeness of the resolution calculus for propositional logic [10].

5 Conclusions

We have developed a sound, complete, and terminating resolution-based proof method for finite-valued logics. In particular, correctness results were easily achieved as they rely on the satisfaction preservation of the normal form based on bivalent semantics, and in syntactical variations of usual resolution-based rules. Also, the fact that a set of formulae \( \Gamma = \{ \gamma_0, \gamma_1, \ldots, \gamma_n \} \) entails \( \varphi \) in the particular many-valued logic under consideration can be translated into the unsatisfiability of \( \land_{i=0}^{n} \tau(T; \gamma_i) \land \tau(F; \varphi) \) in our metalanguage. Hence, strong completeness of our method easily follows from the results offered in the present study.

From our correctness results, a proof in \( \text{RES}_{\mathcal{B}(\mathcal{L}, \overline{\vartheta})} \) may be simulated by ordinary propositional resolution, by using the unfalsifiable formulae originated from the unobtainable partial binary sequences (see Lemma 4.7) and the translation of the set of clauses into the ordinary propositional language, and by then removing labels from metalinguistic formulae and introducing classical meta-negations where appropriate. Also, formulae of the form \( \theta(\varphi) \), where \( \theta \) is a separator and \( \varphi \) is basic,
are not analysable and may be regarded as ground terms, translatable into new atomic symbols. It should thus be clear that automatisation of $\text{RES}_{B(L, \tau)}$ only requires representation into the language of an existing propositional theorem-prover. Implementation of translators from semantic specifications of many-valued logics into propositional logic is left as future work.

The efficiency of the transformation procedure, as compared, for instance, to those of [1] and [6], depends strongly on the size of the $B$-statements, which in turn depends on the set of separating sequences chosen to generate such statements as well as the number and arities of connectives in the object-language. As the transformation of many-valued formulae into singleton-as-signs given in [1], the transformation given here leads to a clause set that is exponential on the size of the original formula. On the bright side, as argued above, the transformation we present produces truly classical formulae in the metalanguage and there is no need to perform unification on labels. We are currently investigating alternative ways to approximate our results to that in [6], where translation is driven by sets-as-signs.

The extension of the above study to first-order logics endowed with distribution quantifiers does not seem so much of a technical challenge, but rather depends on previously extending the underlying reduction algorithm to the first-order case. A more interesting challenge—which we know how to solve, but the solution doesn’t fit here in the margin—is the extension of the above study to cover logics with finite-valued nondeterministic matrices.

References

Computational paths, identity type, and the groupoid model

Arthur F. Ramos
Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil

Ruy J. G. B. de Queiroz
Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil

Anjolina G. de Oliveira
Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil

Abstract

We introduce a new way of formalizing the intensional identity type based on the notion of computational paths which will be taken to be proofs of propositional equality, and thus terms of the identity type. Our approach results in an elimination rule which is much simpler and much more usable than the one given by Martin-Löf in his intensional identity type. In order to make this point clear we construct terms of some relevant types using our proposed elimination rule. We also show that one of the properties of Martin-Löf’s original identity type is present on our formulation of the identity type of computational paths. We are referring to the fact that the identity type induces a groupoid structure, as proposed by Hofmann & Streicher (1994). Using categorical semantics, we show that computational paths induce a groupoid structure too. It is further shown that computational paths induce higher categorical structures.

Keywords: Identity type, computational paths, equality proofs, path-based constructions, type theory, groupoid model, term rewriting systems, category theory, higher categorical structures.

1 Introduction

One interesting peculiarity of Martin-Löf’s Intensional Type Theory is the existence of two distinct kinds of equalities between terms of the same type. The first one

Selected, revised papers will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
is originated by the fact that equality can be seen as a type. The second one is originated by the fact that two terms can be equal by definition.

The treatment of an equality as a type gives rise to an extremely interesting type known as identity type. The idea is that, given terms $a, b$ of a type $A$, one may form the type whose elements are proofs that $a$ and $b$ are equal elements of type $A$. This type is represented by $\text{Id}_A(a, b)$. A term $p : \text{Id}_A(a, b)$ makes up for the *grounds* [15] (or proof) that establishes that $a$ is indeed equal to $b$. We say that $a$ is *propositionally* equal to $b$.

The second kind of equality is called definitional and is denoted by $\equiv$. It occurs when two terms are equal by definition, i.e., there is no need for an evidence or a proof to establish the equality. A classic example, given in [1], is to consider any function definition, for example, $f : \mathbb{N} \to \mathbb{N}$ defined by $f(x) \equiv x^2$. For $x = 3$, we have, by definition, that $f(x) \equiv 3^2$. We are unable to conclude that $f(3) \equiv 9$ though. The problem is that, to conclude that $3^2 = 9$, we need additional evidence. We need a way to compute $3^2$, i.e., we need to use exponentials (or multiplications) to conclude that $3^2$ is equal to $9$. That way, we can only prove that $3^2$ is propositionally equal to $9$, i.e., there is a $p : \text{Id}_\mathbb{N}(3^2, 9)$.

Between those two kind of equalities, the propositional one is, without a doubt, the most interesting one. This claim is based on the fact that many interesting results have been achieved using the identity type. One of these was the discovery of the Univalent Models in 2005 by Vladimir Voevodsky [17]. A groundbreaking result has arisen from Voevdsky’s work: the connection between type theory and homotopy theory. The intuitive connection is simple: a term $a : A$ can be considered as a point of the space $A$ and $p : \text{Id}_A(a, b)$ is a homotopical path between points $a, b \in A$ [1]. This has given rise to a whole new area of research, known as Homotopy Type Theory. It leads to a new perspective on the study of equality, as expressed by Voevodsky in a recent talk in *The Paul Bernays Lectures* (Sept 2014, Zurich): equality (for abstract sets) should be looked at as a *structure* rather than as a *relation*.

Motivated by the fact that the identity type has given rise to such interesting concepts, we have been engaged in a process of revisiting the construction of the intensional identity type, as originally proposed by Martin-Löf. Although beautifully defined, we have noticed that proofs that uses the identity type can be sometimes a little too complex. The elimination rule of the intensional identity type encapsulates lots of information, sometimes making too troublesome the process of finding the reason that builds the correct type.

Inspired by the path-based approach of the homotopic interpretation, we believe that a similar approach can be used to define the identity type in type theory. To achieve that, we have been using a notion of *computational paths*. The interpretation will be similar to the homotopic one: a term $p : \text{Id}_A(a, b)$ will be a computational path between terms $a, b : A$, and such path will be the result of a sequence of rewrites. In the sequel, we shall define formally the concept of a computational path. The main idea, i.e. proofs of equality statements as (reversible) sequences of rewrites, is not new, as it goes back to a paper entitled “Equality in labelled deductive systems and the functional interpretation of propositional equality”, presented in December 1993 at the 9th Amsterdam Colloquium, and published in the proceedings in 1994.
After establishing the connection between the identity type and computational paths, we study important mathematical properties that arise naturally from the concept of computational paths. Specifically, we show that computational paths naturally induce a structure known as a groupoid. In fact, we go a step further, showing that computational paths are capable of inducing higher structures.

## 2 Computational Paths

Since computational path is a generic term, it is important to emphasize the fact that we are using the term computational path in the sense defined by [7]. A computational path is based on the idea that it is possible to formally define when two computational objects $a, b : A$ are equal. These two objects are equal if one can reach $b$ from $a$ applying a sequence of axioms or rules. This sequence of operations forms a path. Since it is between two computational objects, it is said that this path is a computational one. Also, an application of an axiom or a rule transforms (or rewrite) an term in another. For that reason, a computational path is also known as a sequence of rewrites. Nevertheless, before we define formally a computational path, we can take a look at one famous equality theory, the $\lambda \beta \eta$-equality [11]:

**Definition 2.1** The $\lambda \beta \eta$-equality is composed by the following axioms:

- (α) $\lambda x. M = \lambda y. [y/x]M$ if $y \not\in \text{FV}(M)$;
- (β) $(\lambda x. M)N = [N/x]M$;
- (ρ) $M = M$;
- (η) $(\lambda x. Mx) = M$ ($x \not\in \text{FV}(M)$).

And the following rules of inference:

- (μ) $\frac{M = M'}{NM = NM'}$
- (τ) $\frac{M = N}{M = P}$
- (ν) $\frac{M = M'}{MN = M'N}$
- (σ) $\frac{N = M}{N = M}$
- (ξ) $\frac{\lambda x. M = \lambda x. M'}{M = M'}$

**Definition 2.2** ([11]) $P$ is $\beta$-equal or $\beta$-convertible to $Q$ (notation $P =_\beta Q$) if $Q$ is obtained from $P$ by a finite (perhaps empty) series of $\beta$-contractions and reversed $\beta$-contractions and changes of bound variables. That is, $P =_\beta Q$ if there exist $P_0, \ldots, P_n$ ($n \geq 0$) such that $P_0 \equiv P$, $P_n \equiv Q$, $(\forall i \leq n - 1)(P_i \triangleright_\beta P_{i+1}$ or $P_{i+1} \triangleright_\beta P_i$ or $P_i \equiv_{\alpha} P_{i+1}$).

(NB: equality with an existential force, which will show in the proof rules for the identity type.)

The same happens with $\lambda \beta \eta$-equality:
Definition 2.3 (λβη-equality [11]) The equality-relation determined by the theory λβη is called =_βη; that is, we define

\[ M =_βη N \iff \lambdaβη \vdash M = N. \]

Example 2.4 Take the term \( M \equiv (\lambda x.(\lambda y.yx)(\lambda w.zw))v \). Then, it is βη-equal to \( N \equiv zv \) because of the sequence:

\((\lambda x.(\lambda y.yx)(\lambda w.zw))v, (\lambda x.(\lambda y.yx)z)v, (\lambda y.v)z, zv\)

which starts from \( M \) and ends with \( N \), and each member of the sequence is obtained via 1-step β- or η-contraction of a previous term in the sequence. To take this sequence into a path, one has to apply transitivity twice, as we do in the example below.

Example 2.5 The term \( M \equiv (\lambda x.(\lambda y.yx)(\lambda w.zw))v \) is βη-equal to \( N \equiv zv \) because of the sequence:

\((\lambda x.(\lambda y.yx)(\lambda w.zw))v, (\lambda x.(\lambda y.yx)z)v, (\lambda y.v)z, zv\)

Now, taking this sequence into a path leads us to the following:

The first is equal to the second based on the grounds:

\(\eta((\lambda x.(\lambda y.yx)(\lambda w.zw))v, (\lambda x.(\lambda y.yx)z)v)\)

The second is equal to the third based on the grounds:

\(\beta((\lambda x.(\lambda y.yx)z)v, (\lambda y.v)z)\)

Thus, the first one is equal to the fourth one based on the grounds:

\(\tau(\eta((\lambda x.(\lambda y.yx)(\lambda w.zw))v, (\lambda x.(\lambda y.yx)z)v), \beta((\lambda x.(\lambda y.yx)z)v, (\lambda y.v)z))\).

The aforementioned theory establishes the equality between two λ-terms. Since we are working with computational objects as terms of a type, we need to translate the λβη-equality to a suitable equality theory based on Martin Löf’s type theory. We obtain:

Definition 2.6 The equality theory of Martin Löf’s type theory has the following basic proof rules for the Π-type:

\[
\begin{align*}
&\quad \frac{N : A \quad M : B}{(\lambda x . M)N = M[N/x] : B[N/x]} \\
&\quad \frac{M =_\beta M'}{(\lambda x . M) = \lambda x . M'} : (\Pi x : A)B \\
&\quad \frac{M =_\rho M'}{M = M' : A} \\
&\quad \frac{N =_\sigma N'}{N = N' : A} \\
&\quad \frac{M =_\tau N =_\nu P}{M = P : A}
\end{align*}
\]
We are finally able to formally define computational paths:

**Definition 2.7** Let \(a\) and \(b\) be elements of a type \(A\). Then, a **computational path** \(s\) from \(a\) to \(b\) is a composition of rewrites (each rewrite is an application of the inference rules of the equality theory of type theory or is a change of bound variables). We denote that by \(a =_s b\).

As we have seen in example 2.5, composition of rewrites are applications of the rule \(\tau\). Since change of bound variables is possible, consider that each term is considered up to \(\alpha\)-equivalence.

## 3 Identity Type

In this section, we have two main objectives. The first one is to propose a formalization to the identity type using computational paths. The second objective is to show how one can use our approach to construct types representing reflexivity, transitivity and symmetry. In the case of the transitive type, we also compare our approach with the traditional one, i.e., Martin-Löf’s Intensional type. With this comparison, we hope to show the clear advantage of our approach, in terms of simplicity. Since our approach is based on computational paths, we will sometimes refer to our formulation as the path-based approach and the traditional formulation as the pathless approach. By this we mean that, even though the HoTT approach to the identity type brings about the notion of paths in the semantics, there is little in the way of handling paths as terms in the language of type theory.

Before the deductions that build the path-based identity type, we would like to make clear that we will use the following construction of the traditional approach [10]:

\[
\begin{align*}
A \text{ type} & \quad a : A \quad b : A \quad \text{Id} - F \quad a : A \quad r(a) : \text{Id}_A(a,a) \quad \text{Id} - I \\
 a : A & \quad b : A \quad c : \text{Id}_A(a,b) \quad q(x) : C(x,x,r(x)) \quad [x : A] \quad [x : A, y : A, z : \text{Id}_A(x,y)] \\
 & \quad J(p,q) : C(a,b,c) \quad C(x,y,z) \quad \text{type} \quad \text{Id} - E
\end{align*}
\]

### 3.1 The path-based construction

The best way to define any formal entity of type theory is by a set of natural deductions rules. Thus, we define our path-based approach as the following set of rules:

- Formation and Introduction rules:
A type \( a : A \quad b : A \quad \exists a \neq b : A \quad Id \quad s(a, b) : Id_A(a, b) \)  

- Elimination rule:

\[
\begin{align*}
\frac{m : Id_A(a, b) \quad h(g) : C}{REW_R(m, \hat{g}h(g)) : C}
\end{align*}
\]

- Reduction rules:

\[
\begin{align*}
\frac{a =_m b : A \quad m(a, b) : Id_A(a, b)}{REW_R(m, \hat{g}h(g)) : C}
\end{align*}
\]

In these rules, \( \hat{g} \) (and \( \hat{t} \)) to indicate that they are abstractions over the variable \( g \) (or \( t \)), for which the main rules of conversion of \( \lambda \)-abstraction hold. For that reason, we proposed two reduction rules that handle these conversions, the \( \beta \) and \( \eta \) reduction rules.

3.2 Path-based constructions

The objective of this subsection is to show how to use in practice the rules that we have just defined. The idea is to show construction of terms of some important types. The constructions that we have chosen to build are the reflexive, transitive and symmetric type of the identity type. Those were not random choices. The main reason is the fact that reflexive, transitive and symmetric types are essential to the process of building a groupoid model for the identity type [12]. As we shall see, these constructions come naturally from simple computational paths constructed by the application of axioms of the equality of type theory. In contrast, we will also show that constructing the transitivity using the operator \( J \) can be a little complicated. With that, we hope to make the simplicity and advantage of our approach clear.

Before we start the constructions, we think that it is essential to understand how to use the eliminations rules. The process of building a term of some type is a matter of finding the right reason. In the case of \( J \), the reason is the correct \( x, y : A \) and \( z : Id_A(a, b) \) that generates the adequate \( C(x, y, z) \). In our approach, the reason is the correct path \( a =_g b \) that generates the adequate \( g(a, b) : Id(a, b) \).

One could find strange the fact that we need to prove the reflexivity. Nevertheless, just remember that our approach is not based on the idea that reflexivity is
the base of the identity type. As usual in type theory, a proof of something comes down to a construction of a term of a type. In this case, we need to construct a
term of type \( \Pi_{(a:A)} \text{Id}_A(a, a) \). The reason is extremely simple: from a term \( a : A \),
we obtain the computational path \( a =_\rho a : A \):

\[
\begin{align*}
[a : A] & \implies \frac{a =_\rho a : A}{\frac{\rho(a,a) : \Pi_{(a:A)} \text{Id}_A(a,a)}{\lambda a. \rho(a,a) : \Pi_{(a:A)} \text{Id}_A(a,a)}} \Pi - I
\end{align*}
\]

The second proposed construction is the symmetry. We need to construct a term
of type \( \Pi_{(a:A)} \Pi_{(b:A)} (\text{Id}_A(a, b) \rightarrow \text{Id}_A(b, a)) \). As expected, we need to find a suitable reason. Starting from \( a =_t b \), we could look at the axioms of definition 2.3 to plan our next step. One of those axioms makes the symmetry clear: the \( \sigma \) axiom. If we apply \( \sigma \), we will obtain \( b =_{\sigma (t)} a \). From this, we can then infer that \( \text{Id}_A \) is inhabited by \( (\sigma(t))(b, a) \). Now, it is just a matter of applying the elimination:

\[
\begin{align*}
[a : A] & \frac{[b : A]}{\frac{\rho(a,b) : \text{Id}_A(a,b)}{(\sigma(t))(b,a) : \text{Id}_A(b,a)}} \frac{\text{Id} - I}{\text{Id} - E} \\
\frac{\text{REWR} \{ (p(a,b), \lambda (t). (\sigma(t))(b,a)) : \text{Id}_A(a,b) \rightarrow \text{Id}_A(b,a) \}}{\text{REWR} \{ (p(a,b), \lambda (t). (\sigma(t))(b,a)) : \text{Id}_A(a,b) \rightarrow \text{Id}_A(b,a) \}} \frac{\Pi - I}{\Pi - I}
\end{align*}
\]

The third and last construction will be the transitivity. The transitivity will be a
special case, since we will also show the construction using the pathless approach, i.e.,
using the operator \( J \). The objective is to show the difference of complexity in the
process of finding a suitable reason. Both approaches have the objective of constructing
a term for the type \( \Pi_{(a:A)} \Pi_{(b:A)} \Pi_{(c:A)} (\text{Id}_A(a,b) \rightarrow \text{Id}_A(b,c) \rightarrow \text{Id}_A(c,a)) \).

Let’s start with the construction based on \( J \). The proof will be based on the one found in [1]. The difference is that instead of defining induction principles for \( J \) based on the elimination rules, we will use the rule directly. The complexity is the same, since the proofs are two forms of presenting the same thing and they share the same reasons. As one should expect, the first and main step is to find a suitable reason. In other words, we need to find suitable \( x, y : A \) and \( z : \text{Id}_A(a,b) \) to construct an adequate \( C(x,y,z) \). This first step is already problematic. Different from our approach, where one starts from a path and applies intuitive equality axioms to find a suitable reason, there is no clear point of how one should proceed to find a suitable reason for the construction based on \( J \). In this case, one should rely on intuition and make attempts until one finds out the correct reason. As one can check in [1],
a suitable reason would be \( x : A, y : A, - : \text{Id}_A(x,y) \) and \( C(x,y,z) = \text{Id}_A(y,c) \rightarrow \text{Id}_A(x,c) \). The symbol \( - \) indicates that \( z \) can be anything, i.e., the choice of \( z \) will

157
not matter. Looking closely, the proof is not over yet. The problem is the type of \( C(x, x, r(x)) \). With this reason, we have that \( C(x, x, r(x)) \equiv I_d A(x, c) \to I_d A(x, c) \). Therefore, we cannot assume that \( q(x) : I_d A(x, c) \to I_d A(x, c) \) is the term \( r(x) \). The only way to proceed is to apply again the constructor \( J \) to build the term \( q(x) \). It means, of course, that we will need to find yet another reason to build this type.

This second reason is given by \( x : A, y : A, - : I_d A(x, y) \) and \( C'(x, y, z) \equiv I_d A(x, y) \). In that case, \( C'(x, x, r(x)) = I_d A(x, x) \). We will not need to use \( J \) again, since now we have that \( r(x) : I_d A(x, x) \). Then, we can construct \( q(x) \):

\[
\begin{array}{c}
\frac{x : A \quad c : A \quad q : I_d A(x, c) \quad r(x) : I_d A(x, x)}{\lambda q.J(q, r(x)) : I_d A(x, c) \to I_d A(x, c)}
\end{array}
\]

We can finally obtain the desired term:

\[
\begin{array}{c}
\frac{a : A \quad b : A \quad p : I_d A(a, b) \quad \lambda q.J(q, r(x)) : I_d A(x, c) \to I_d A(x, c)}{\lambda p.J(p, \lambda q.J(q, r(x))) : I_d A(b, c) \to I_d A(a, c)}
\end{array}
\]

This construction is an example that makes clear the difficulties of working with the pathless formulation. We had to find two different reasons and use two applications of the elimination rule. Another problem is the fact that the reasons were not obtained by a fixed process, like the applications of axioms in some entity of type theory. They were obtained purely by the intuition that a certain \( C(x, y, z) \) should be capable of constructing the desired term. For that reason, obtaining these reasons can be troublesome.

We finish our constructions by giving the path-based construction of the transitivity. The first step, as expected, is to find the reason. Since we are trying to construct the transitivity, it is natural to think that we should start with paths \( a =_l b \) and \( b =_u c \) and then, from these paths, we should conclude that there is a path \( z \) that establishes that \( a =_z c \). To obtain \( z \), we could try to apply the axioms of definition 2.3. Looking at the axioms, one is exactly what we want: the axiom \( \tau \). If we apply \( \tau \) to \( a =_l b \) and \( b =_u c \), we will obtain a new path \( \tau(t, u) \) such that \( a =_\tau(t, u) c \). Using that construction as the reason, we obtain the following term:
As one can see, each step is just straightforward applications of introduction, elimination rules and abstractions. The only idea behind this construction is just the simple fact that the axiom $\tau$ guarantees the transitivity of paths. If one compares the reason of this construction to the one that used $J$, one can clearly conclude that the reason of the path-based approach was obtained more naturally and more concretely.

### 4 A Term Rewriting System for Paths

As we have just showed, a computational path establishes when two terms of the same type are equal. From the theory of computational paths, an interesting case arises. Suppose we have a path $s$ that establishes that $a =_s b : A$ and a path $t$ that establishes that $a =_t b : A$. Consider that $s$ and $t$ are formed by distinct compositions of rewrites. Is it possible to conclude that there are cases that $s$ and $t$ should be considered equivalent? The answer is yes. Consider the following example:

**Example 4.1** Consider the path $a =_t b : A$. By the symmetric property, we obtain $b =_{\sigma(t)} a : A$. What if we apply the property again on the path $\sigma(t)$? We would obtain a path $a =_{\sigma(\sigma(t))} b : A$. Since we applied symmetry twice in succession, we obtained a path that is equivalent to the initial path $t$. For that reason, we conclude the path of applying symmetry twice in succession is a redundancy. We say that the path $\sigma(\sigma(t))$ can be reduced to the path $t$.

As one could see in the aforementioned example, different paths should be considered equal if one is just a redundant form of the other. The example that we have just seen is just a straightforward and simple case. Since the equality theory has a total of 7 axioms, the possibility of combinations that could generate redundancies are high. Fortunately, all possible redundancies were thoroughly mapped by De Oliveira (1995)[3]. In this work, a system that establishes all redundancies and creates rules that solve them was proposed. This system, known as $LND_{EQ} - TRS$, maps a total of 39 rules that solve redundancies. Since we have a special interest on the groupoid model, we are not interested in all redundancy rules, but in a very specific subset of these rules (all have been taken from [3,6]):

- Rules involving $\sigma$ and $\rho$

\[
\frac{x =_{\rho} x : A}{x =_{\sigma(\rho)} x : A} \quad \triangleright_{sr} \quad x =_{\rho} x : A
\]
\[
\frac{x = y : A}{y = \sigma(x) : A} \quad x = \sigma(y) : A
\]

• Rules involving \(\tau\)

\[
\frac{x = y : A \quad y = \sigma(r) : A}{x = \tau(r, \sigma(r)) : A} \quad \triangleright \quad x = \tau(x) : A
\]

\[
\frac{y = \sigma(r) : A \quad x = y : A}{y = \tau(r, \sigma(r)) : A} \quad \triangleright \quad y = \tau(y) : A
\]

\[
\frac{x = y : A \quad y = \tau(r, \sigma(r)) : A}{x = \tau(r, \sigma(r)) : A} \quad \triangleright \quad x = \tau(x) : A
\]

• Rule involving \(\tau\) and \(\tau\)

\[
\frac{x = t : A \quad y = w : A}{x = \tau(t, r) : A} \quad w = z : A
\]

\[
\frac{x = \tau(t, r) : A \quad w = z : A}{x = \tau(t, \sigma(r)) : A} \quad \triangleright \quad x = \tau(x) : A
\]

\[
\frac{y = w : A \quad w = z : A}{y = \tau(r, s) : A} \quad \triangleright \quad y = \tau(y) : A
\]

\[
\frac{x = \tau(t, r) : A \quad y = \tau(r, s) : A}{x = \tau(t, \sigma(r)) : A} \quad \triangleright \quad x = \tau(x) : A
\]

Definition 4.2 A \(rw\)-rule is any of the rules defined in \(LND_{EQ} - TRS\).

Definition 4.3 Let \(s\) and \(t\) be computational paths. We say that \(s \triangleright w\) (read as: \(s\) \(rw\)-contracts to \(t\)) if \(t\) can be obtained from \(s\) by an application of only one \(rw\)-rule. If \(s\) can be reduced to \(t\) by finite number of \(rw\)-contractions, then we say that \(s \triangleright w\) (read as \(s\) \(rw\)-reduces to \(t\)).

Definition 4.4 Let \(s\) and \(t\) be computational paths. We say that \(s = w\) (read as: \(s\) is \(rw\)-equal to \(t\)) if \(t\) can be obtained from \(s\) by a finite (perhaps empty) series of \(rw\)-contractions and reversed \(rw\)-contractions. In other words, \(s = w\) if there exists a sequence \(R_0, ..., R_n\), with \(n \geq 0\), such that

\[
(\forall i \leq n - 1)(R_i \triangleright w = s \text{ and } R_{i+1} \triangleright w = t)
\]

Proposition 4.5 \(rw\)-equality is transitive, symmetric and reflexive.

Proof. Comes directly from the fact that \(rw\)-equality is the transitive, reflexive and symmetric closure of \(rw\).

Before talking about higher \(LND_{EQ} - TRS\) systems, we'd like to mention that \(LND_{EQ} - TRS\) is terminating and confluent. The proof of this affirmation can be found in [3,4,5,8].

160
4.1 \textit{LND}_{\textit{EQ}} − \textit{TRS}_2

Until now, this section has concluded that there exist redundancies which are resolved by a system called \textit{LND}_{\textit{EQ}} − \textit{TRS}. This system establishes rules that reduces these redundancies. Moreover, we concluded that these redundancies are just redundant uses of the equality axioms showed in section 2. In fact, since these axioms just defines an equality theory for type theory, one can specify and say that these are redundancies of the equality of type theory. As we mentioned, the \textit{LND}_{\textit{EQ}} − \textit{TRS} has a total of 39 rules \cite{3}. Since the \textit{rw}-equality is based on the rules of \textit{LND}_{\textit{EQ}} − \textit{TRS}, one can just imagine the high number of redundancies that \textit{rw}-equality could cause. In fact, a thoroughly study of all the redundancies caused by these rules could generate an entire new work. Fortunately, we are only interested in the redundancies caused by the fact that \textit{rw}-equality is transitive, reflexive and symmetric with the addition of only one specific \textit{rw}-rule. Let’s say that we have a system, called \textit{LND}_{\textit{EQ}} − \textit{TRS}_2, that resolves all the redundancies caused by \textit{rw}-equality (the same way that \textit{LND}_{\textit{EQ}} − \textit{TRS} resolves all the redundancies caused by equality). Since we know that \textit{rw}-equality is transitive, symmetric and reflexive, it should have the same redundancies that the equality had involving only these properties. Since \textit{rw}-equality is just a sequence of \textit{rw}-rules (also similar to equality, since equality is just a computational path, i.e., a sequence of identifiers), then we could put a name on these sequences. For example, if \( s \) and \( t \) are \textit{rw}-equal because there exists a sequence \( \theta : R_0, \ldots, R_n \) that justifies the \textit{rw}-equality, then we can write that \( s =_{\textit{rw}_0} t \). Thus, we can rewrite, using \textit{rw}-equality, all the rules that originated the rules involving \( \tau, \sigma \) and \( \rho \). For example, we have:

\[
\begin{align*}
x =_{\textit{rw}_1} y : A & \quad y =_{\textit{rw}_1} w : A \\
x =_{\textit{rw}_2} w : A & \quad w =_{\textit{rw}_3} z : A \\
x =_{\textit{rw}_3(r(r, s))} z : A
\end{align*}
\]

\[
\begin{align*}
\triangleright_{\textit{tt}_2} x =_{\textit{rw}_4} y : A & \quad y =_{\textit{rw}_4} w : A \\
& \quad w =_{\textit{rw}_5} z : A \\
x =_{\textit{rw}_6(r(r, s))} z : A
\end{align*}
\]

Therefore, we obtain the rule \( \textit{tt}_2 \), that resolves one of the redundancies caused by the transitivity of \textit{rw}-equality (the 2 in \( \textit{tt}_2 \) indicates that it is a rule that resolves a redundancy of \textit{rw}-equality). In fact, using the same reasoning, we can obtain, for \textit{rw}-equality, all the redundancies that we have showed in page 5. In other words, we have \( \textit{tr}_2, \textit{tsr}_2, \textit{ttr}_2, \textit{tlr}_2, \textit{sr}_2, \textit{ss}_2 \) and \( \textit{tt}_2 \). Since we have now rules of \textit{LND}_{\textit{EQ}} − \textit{TRS}_2, we can use all the concepts that we have just defined for \textit{LND}_{\textit{EQ}} − \textit{TRS}. The only difference is that instead of having \textit{rw}-rules and \textit{rw}-equality, we have \textit{rw}_2-rules and \textit{rw}_2-equality.

The rule specific to this system is one originated from the fact that the transitivity of reducible paths can be reduced in different ways, but generating the same result. For example, consider the simple case of \( \tau(s, t) \) and consider that it is possible to reduce \( s \) to \( s' \) and \( t \) to \( t' \). There is two possible \textit{rw}-sequences that reduces this case: The first one is \( \theta : \tau(s, t) \triangleright_{\textit{1rw}} \tau(s', t) \triangleright_{\textit{1rw}} \tau(s', t') \) and the second
Both \( \tau(s,t) \) \( 1 \text{rw} \) \( \tau(s,t') \) \( 1 \text{rw} \) \( \tau(s',t') \). Both \( \text{rw} \)-sequences obtained the same result in similar ways, the only difference being the choices that have been made at each step. Since the variables, when considered individually, followed the same reductions, these \( \text{rw} \)-sequences should be considered redundant relative to each other and, for that reason, there should be \( \text{rw} \)-rule that establishes this reduction. This rule is called \textit{choice independence} and is denoted by \( \text{cd}_2 \). In fact, independent of the quantity of transitivities and variables, if the sole difference between the \( \text{rw} \)-sequences are the choices that were made in each step, then this rule will establish the \( \text{rw} \)-equality between the sequences.

**Proposition 4.6** \( \text{rw} \)-equality is transitive, symmetric and reflexive.

**Proof.** Analogous to Proposition 4.5.

5 The Groupoid Induced by Computational Paths

The objective of this section is to show that computational paths, together with the reduction rules discussed in the last section, are capable of inducing structures known as groupoids. To do that, we will use a categorical interpretation.

Before we conclude that computational paths induces categories with groupoid properties, we need to make clear the difference between a strict and a weak category. As one will see, the word weak will appear many times. This will be the case because some of the categorical equalities will not hold “on the nose”, so to say. They will hold up to \( \text{rw} \)-equality or up to higher levels of \( \text{rw} \)-equalities. This is similar to the groupoid model of the identity type proposed by [12]. In [12], the equalities do not hold “on the nose”, they hold up to propositional equality (or up to homotopy if one uses the homotopic interpretation). To indicate that these equalities hold only up to some property, we say that the induced structure is a weak categorical structure.

For each type \( A \), computational paths induces a weak categorical structure called \( A_{\text{rw}} \). In \( A_{\text{rw}} \), the objects are terms \( a \) of the type \( A \) and morphisms between terms \( a : A \) and \( b : A \) are arrows \( s : a \to b \) such that \( s \) is a computational path between the terms, i.e., \( a =_s b : A \). One can easily check that \( A_{\text{rw}} \) is a category. To do that, just define the composition of morphisms as the transitivity of two paths and the reflexive path as the identity arrow. The \( \text{rw} \)-equality of the associativity comes directly from the \( tt \) rule and the \( \text{rw} \)-equality of the identity laws comes from the \( trr \) and \( tlr \) rules. Of course, as discussed before, this only forms a weak structure, since the equalities only hold up to \( \text{rw} \)-equality. The most interesting fact about \( A_{\text{rw}} \) is the following proposition:

**Proposition 5.1** The induced structure \( A_{\text{rw}} \) has a weak groupoidal structure.

**Proof.** A groupoid is just a category in which every arrow is an isomorphism. Since we are working in a weak sense, the isomorphism equalities need only to hold up to \( \text{rw} \)-equality. To show that, for every arrow \( s : a \to b \), we need to show a \( t : b \to a \) such that \( t \circ s =_{\text{rw}} 1_a \) and \( s \circ t =_{\text{rw}} 1_b \). To do that, recall that every computational path has an inverse path \( \sigma(s) \). Put \( t = \sigma(s) \). Thus, we have that \( s \circ t = s \circ \sigma(s) = \tau(\sigma(s),s) \triangleright_{\text{tsr}} \rho_b \). Since \( \rho_b = 1_b \), we conclude that \( s \circ t =_{\text{rw}} 1_b \).
We also have that $t \circ s = \sigma(s) \circ s = \tau(s, \sigma(s)) \triangleright tr \rho_a$. Therefore, $t \circ s = 1_a$. We conclude that every arrow is a weak isomorphism and thus, $A_{rw}$ is a weak groupoidal structure. 

With that, we conclude that computational paths have a groupoid model.

### 5.1 Higher structures

We have just showed that computational paths induce a weak groupoidal structure known as $A_{rw}$. We also know that the arrows (or morphisms) of $A_{rw}$ are computational paths between two terms of a type $A$. As we saw in the previous section, sometimes a computational path can be reduced to another by rules that we called $rw$-rules. That way, if we have terms $a, b : A$ and paths between these terms, we can define a new structure. This new structure, called $A_{2rw}(a, b)$, has, as objects, computational paths between $a$ and $b$ and the set of morphisms $Hom_{A_{2rw}(a, b)}(s, t)$ between paths $a =_s b$ and $a =_t b$ corresponds to the set of sequences that prove $s =_{rw} t$. Since $rw$-equality is transitive, reflexive and symmetric, $A_{2rw}$ is a weak categorical structure which the equalities hold up to $rw_2$-equality. The proof of this fact is analogous to the proposition 4.1. The sole difference is the fact that since the morphisms are $rw$-equalities, instead of computational paths, all the equalities will hold up to $rw_2$-equality. To see this, take the example of the associativity. Looking at the $LND_{EQ} - TRS_2$ system, we have that $\tau(\tau(\theta, \sigma, \phi)) \triangleright_{tr} \tau(\theta, \tau(\sigma, \phi)) \triangleright \theta, \sigma$ and $\phi$ represent $rw$-equalities between paths from $a$ to $b$. Therefore, $\tau(\tau(\theta, \sigma, \phi)) =_{rw_2} \tau(\theta, \tau(\sigma, \phi))$. The associative law holds up to $rw_2$-equality. As one can easily check, the identity law will also hold up to $rw_2$-equality. Therefore, $A_{2rw}(a, b)$ has a weak categorical structure. Analogous to proposition 4.4, the groupoid law will also hold up to $rw_2$-equality.

Instead of considering $A_{rw}$ and $A_{2rw}(a, b)$ as separated structures, we can think of a unique structure with 2-levels. The first level is $A_{rw}$ and the second one is the structure $A_{2rw}(a, b)$ between each pair of objects $a, b \in A_{rw}$. We call this structure $2 - A_{rw}$. The morphisms of the first level are called 1-morphisms and the ones of the second level are called 2-morphisms (also known as 2-arrows or 2-cells). Since it has multiple levels, it is considered a higher structure. We want to prove that this structure is a categorical structure known as weak 2-category. The main problem is the fact that in a weak 2-category, the last level (i.e., the second level) needs to hold up in a strict sense. This is not the case for $2 - A_{2rw}$, since each $A_{2rw}(a, b)$ only holds up to $rw_2$-equality. Nevertheless, there still a way to induce this weak 2-category. Since $rw$-equality is an equivalence relation (because it is transitive, symmetric and reflexive), we can consider a special $A_{2rw}(a, b)$, where the arrows are the arrows of $A_{2rw}(a, b)$ modulo $rw_2$-equality. That way, since the equalities hold up to $rw_2$-equality in $A_{2rw}(a, b)$, they will hold in a strict sense when we consider the equivalence classes of $rw_2$-equality. We call this structure $[A_{2rw}(a, b)]$. In this structure, consider the composition of arrows defined as: $[\theta]_{rw_2} \circ [\phi]_{rw_2} = [\theta \circ \phi]_{rw_2}$. Now, we can think of the structure $[2 - A_{rw}]$. This structure is similar to $2 - A_{rw}$. The difference is that the categories of the second level are $[A_{2rw}(a, b)]$ instead of $A_{2rw}(a, b)$. We can now prove that $[2 - A_{rw}]$ is a weak 2-category:

**Proposition 5.2** Computational paths induce a weak 2-category called $[2 - A_{rw}]$. 

163
Proof. First of all, let’s draw a diagram that represents $[2 - Arw]$:

![Diagram](https://via.placeholder.com/150)

In this diagram we represent 1-arrows and 2-arrows between these 1-arrows. The fact that 2-arrows are equivalence classes is represented by the brackets.

Given $[\alpha]_{rw_2} : [s = \alpha_1, ..., \alpha_n = t]$ and $[\theta]_{rw_2} : [r = \theta_1, ..., \theta_m = w]$, then we define the horizontal composition $([\theta]_{rw_2} \circ_h [\alpha]_{rw_2})$ as the sequence $[\tau(s = \alpha_1, r = \theta_1), ..., \tau(\alpha_n, \theta_1), ..., \tau(\alpha_n = t, \theta m = w)]_{rw_2}$.

We also need to verify the associative and identity law for $\circ_h$. Since we are working with a weak 2-category, these laws should hold up to natural isomorphism [13]. To verify these laws, the idea is that every 2-morphism of $[Arw_{2}]$ is an isomorphism. The proof of this fact is analogous to the one of Proposition 5.1, but using $rw_2$-rules instead of $rw$-rules. Since a natural transformation is a natural isomorphism iff every component is an isomorphism (as one can check in [2]), we conclude that finding isomorphisms for the associative and identity laws is just a matter of finding the correct morphisms.

For the associative law, we need to check that there is a natural isomorphism assoc between $([\psi]_{rw_2} \circ_h [\theta]_{rw_2}) \circ_h [\alpha]_{rw_2}$ and $([\psi]_{rw_2} \circ_h ([\theta]_{rw_2} \circ_h [\alpha]_{rw_2}))$. To do this, by the definition of horizontal composition, a component of $([\psi]_{rw_2} \circ_h [\theta]_{rw_2})$ is a term of the form $\tau(\alpha_x, \tau(\theta_y, \psi_z))$, with $x, y$, and $z$ being suitable natural numbers that respect the order of the horizontal composition. Analogously, the same component of $([\psi]_{rw_2} \circ_h ([\theta]_{rw_2} \circ_h [\alpha]_{rw_2}))$ is just a suitable term $\tau(\tau(\alpha_x, \theta_y), \psi_z)$.

The isomorphism between these component is clearly established by the inverse $tt$ rule, i.e., $\tau(\alpha_x, \tau(\theta_y, \psi_z)) =_{rw_{e(tl)}} \tau(\tau(\alpha_x, \theta_y), \psi_z)$

The identity laws use the same idea. We need to check that $([\alpha]_{rw_2} \circ_h [\rho_{\rho_a}]_{rw_2}) = [\alpha]_{rw_2}$. To do that, we need to take components $(\rho_{\rho_a}, \alpha_y)$ and $\alpha_y$ and establish their isomorphism $r^*_s$: $(\rho_{\rho_a}, \alpha_y) =_{rw_{e(tl)}} \alpha_y$.

The other natural isomorphism $l^*_s$, i.e., the isomorphism between $([\rho_{\rho_a}]_{rw_2} \circ_h [\alpha]_{rw_2})$ and $[\alpha]_{rw_2}$ can be established in an analogous way, just using the rule $tt$ instead of $tl$. Just for purpose of clarification, $\rho_{\rho_a}$ comes from the reflexive property of $rw$-equality. Since $\rho_a$ is the identity path, using the reflexivity we establish that $\rho_a =_{rw} \rho_a$, generating $\rho_{\rho_a}$.

With the associative and identity isomorphisms established, we now need to check the interchange law [13]. We need to check that:

$([\varphi]_{rw_2} \circ [\theta]_{rw_2}) \circ_h ([\chi]_{rw_2} \circ [\alpha]_{rw_2}) = ([\varphi]_{rw_2} \circ_h [\chi]_{rw_2}) \circ ([\theta]_{rw_2} \circ_h [\alpha]_{rw_2})$

From $([\varphi]_{rw_2} \circ [\theta]_{rw_2}) \circ_h ([\chi]_{rw_2} \circ [\alpha]_{rw_2})$, we have:

$([\varphi]_{rw_2} \circ [\theta]_{rw_2}) \circ_h ([\chi]_{rw_2} \circ [\alpha]_{rw_2}) = \tau(\theta, \varphi)_{rw_2} \circ_h [\tau(\alpha, \chi)]_{rw_2} = \theta_1, ..., \theta_n = \varphi_1, ..., \varphi_{n'}]_{rw_2} \circ_h ([\alpha_1, ..., \alpha_m = \chi_1, ..., \chi_{m'}]_{rw_2} =}$

164
[τ(α₁,θ₁),...,τ(αₘ = χ₁,θ₁),...,τ(χₙ,θ₁),...,τ(χₙ,θₘ′ = ϕ₁),...,τ...(165)

From (([ϕ]rw₂ ◦ h [χ]rw₂) ◦ ([θ]rw₂ ◦ h [α]rw₂)):

([τ(α₁,θ₁),...,τ(αₘ,θₘ′),...,τ(χ₁,ϕ₁),...,τ(χₙ,ϕₙ′)]rw₂ ◦

[τ(α₁,θ₁),...,τ(αₘ,θₘ′),...,τ(χ₁,ϕ₁),...,τ(χₙ,ϕₙ′)]rw₂ =

If one looks closely, one can notice that this is a suitable to apply cd₂. Individually, every variable that appears in the sequence of transitivity follows the same expansion in both cases. The only difference is how the choices have been made. Therefore, the rw₂-equality is established by cd₂. Since we are working with equivalence classes, this equality holds strictly.

About the coherence laws, one can easily check that the pentagon and triangle diagrams are commutative by simple and straightforward applications of the natural isomorphisms.

We can also conclude that [2 − Arw] has a weak 2-groupoid structure. That is the case because we already know (from proposition 4.4) that the groupoid laws are satisfied by 1-morphisms up to the isomorphism of the next level, i.e., up to rw-equality and the 2-morphisms, as we have just seen, are isomorphisms (that hold in a strict way, since the second level is using classes of equivalence).

Our objective in a future work is to define higher levels of rw-equalities and, from that, we will try to obtain even higher groupoid structures. Eventually, our goal will be the construction of a weak ω-groupoid. We believe that it is possible to achieve these results, since it was proved by [14,16] that the identity type induces such structure. Given the connection between computational paths and terms of identity type, we should be able to prove that computational paths also induces a weak ω-groupoid.

6 Conclusion

Inspired by a recent discovery that the propositional equality between terms can be interpreted as the type of homotopical paths, we have revisited the formulation of the intensional identity type, proposing a new approach based on an entity known as computational path. We have proposed that a computational path a =sb : A gives grounds to building a term s(a, b) of the identity type, i.e., s(a, b) : Idₐ(a, b), and is formed by a composition of basic rewrites, each with their identifiers taken as constants. We have also developed our approach, showing how the path-based identity type can be rather straightforwardly used in deductions. In particular, we have shown the simplicity of our elimination rule, demonstrating that it is based on path constructions, which are built from applications of simple axioms of the equality for type theory. To make our point even clearer, we have exposed three path-based constructions. More specifically, constructions that prove the transitivity, reflexivity and symmetry of the propositional equality. We have also argued that, in our approach, the process of finding the reason that allows for building the desired term is usually simple and straightforward. At the same time, in the
Ramos, de Queiroz, and de Oliveira

traditional (or pathless) approach, this is not entirely true, since finding the correct reason can be a cumbersome process.

After establishing the foundations of our approach, we analyzed one important structure that the traditional identity type induces: the algebraic structure known as groupoid. Our objective was to show that our approach is on a par with the pathless one, i.e., our path-based identity also induces a groupoid structure. To prove that, we have shown that the axioms of equality generate redundancies, which are resolved by paths between paths. We mentioned that there already exists a system called $LND_{EQ} - TRS$ that maps and resolves these redundancies. We have gone further, proposing the existence of a higher $LND_{EQ} - TRS_2$ which resolves redundancies generated by the $LND_{EQ} - TRS$ system. Using $LND_{EQ} - TRS$, we have proved that a computational path is capable of inducing a weak groupoid structure. Using the higher rewriting system, we have induced a structure known as weak 2-groupoid. With that, we believe we have opened the way, in a future work, for a possible proof establishing that computational paths induces a weak $\omega$-groupoid.

References

Formalization of simplification for context-free grammars

Marcus V. M. Ramos 1
Centro de Informática
UFPE
Recife, Brazil

Ruy J. G. B. de Queiroz 2
Centro de Informática
UFPE
Recife, Brazil

Abstract

Context-free grammar simplification is a subject of high importance in computer language processing technology as well as in formal language theory. This paper presents a formalization, using the Coq proof assistant, of the fact that general context-free grammars generate languages that can be also generated by simpler and equivalent context-free grammars. Namely, useless symbol elimination, inaccessible symbol elimination, unit rules elimination and empty rules elimination operations were described and proven correct with respect to the preservation of the language generated by the original grammar.

Keywords: Context-free language theory, context-free grammars, grammar simplification, useless symbol elimination, inaccessible symbol elimination, empty rule elimination, unit rule elimination, formalization, formal mathematics, proof assistant, interactive proof systems, program verification, Coq.

1 Introduction

The formalization of context-free language theory is key to the certification of compilers and programs, as well as to the development of new languages and tools for certified programming. The results presented in this paper are part of an ongoing work that intends to formalize parts of the context-free language theory in the Coq proof assistant. The initial results comprised the formalization of closure properties for context-free grammars, namely union, concatenation and Kleene star [30].

In order to follow this paper, the reader is required to have basic knowledge of Coq and of context-free language theory. For the beginner, the recommended starting points for Coq are the book by Bertot [7], the online book by Pierce [15].

1 Email:mvmr@cin.ufpe.br
2 Email:ruy@cin.ufpe.br

Selected, revised papers will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
and a few tutorials available on [20]. More information on the Coq proof assistant, as well as on the syntax and semantics of the following definitions and statements, can be found in [12]. Background on context-free language theory can be found in [33], [19] or [31], among others.

The objective of this work is to formalize a substantial part of context-free language theory in the Coq proof assistant, making it possible to reason about it in a fully checked environment, with all the related advantages. Initially, however, the focus has been restricted to context-free grammars and associated results. Pushdown automata and their relation to context-free grammars shall be considered in the future.

When the work is complete, it should be useful for a few different purposes. Among them, to make available a complete and mathematically precise description of the behavior of the objects of context-free language theory. Second, to offer fully checked and mechanized demonstrations of its main results. Third, to provide a library with basic and fundamental lemmas and theorems about context-free grammars and derivations that can be used as a starting point to prove new theorems and increase the amount of formalization for context-free language theory. Fourth, to allow for the certified and efficient implementation of its relevant algorithms in a programming language. Fifth, to permit the experimentation in an educational environment in the form of a tool set, in a laboratory where further practical observations and developments can be done, for the benefit of students, teachers, professionals and researchers.

The general idea of formalizing context-free language theory in the Coq proof assistant is discussed in Section 2. Specific results related to the formalization of grammar simplification are presented in Section 3. The plan for the rest of this research is presented in Section 4 and in Section 5 related work by various other researchers is considered.

The results reported in this paper are related to the elimination of symbols (terminals and non-terminals) in context-free grammars that do not contribute to the language being generated, and also to the elimination of unit and empty rules, in order to shorten the derivation of the sentences of the language.

All the definitions and proof scripts presented in this paper were written in plain Coq and are available for download at: https://github.com/mvmramos/simplification

2 Basic Definitions

Context-free grammars were represented in Coq very closely to the usual algebraic definition $G = (V, \Sigma, P, S)$, where $V$ is the vocabulary of $G$ (it includes all non-terminal and terminal symbols), $\Sigma$ is the set of terminal symbols (used in the construction of the sentences of the language generated by the grammar), $N = V \setminus \Sigma$ is the set of non-terminal symbols (representing different sentence abstractions), $P$ is the set of rules and $S \in N$ is the start symbol (also called initial or root symbol). Rules have the form $\alpha \to \beta$, with $\alpha \in N$ and $\beta \in V^*$. Basic definitions in Coq are presented below. The $N$ and $\Sigma$ sets are represented.
separately from \( G \) (respectively by types \texttt{non\_terminal} and \texttt{terminal}). Notations \texttt{sf} (sentential form) and \texttt{sentence} represent lists, possibly empty, of respectively terminal and non-terminal symbols and terminal only symbols.

\begin{verbatim}
Variables non_terminal terminal : Type.
Notation sf := (list (non_terminal + terminal)).
Notation sentence := (list terminal).
Notation nlist:= (list non_terminal).
\end{verbatim}

The record representation \texttt{cfg} has been used for \( G \). The definition states that \texttt{cfg} is a new type and contains three components. The first is the \texttt{start\_symbol} of the grammar (a non-terminal symbol) and the second is \texttt{rules}, that represent the rules of the grammar. Rules are propositions (represented in Coq by \texttt{Prop}) that take as arguments a non-terminal symbol and a (possibly empty) list of non-terminal and terminal symbols (corresponding, respectively, to the left and right-hand side of a rule).

The predicate \texttt{rules\_finite\_def} assures that the set of rules of the grammar is finite by proving that the length of right-hand side of every rule is equal or less than a given value, and also that both left and right-hand side of the rules are built from finite sets of, respectively, non-terminal and terminal symbols (represented here by lists).

\begin{verbatim}
Definition rules\_finite\_def (ss: non_terminal)
  (rules: non_terminal -> sf -> Prop)
  (n: nat)
  (ntl: list non_terminal)
  (tl: list terminal) :=
In ss ntl /\ 
  (forall left: non_terminal,
   forall right: list (non_terminal + terminal),
   rules left right ->
     length right <= n /\ 
   In left ntl /\ 
   (forall s : non_terminal, In (inl s) right -> In s ntl) /\ 
   (forall s : terminal, In (inr s) right -> In s tl)).
\end{verbatim}

The decision of representing rules as propositions has the consequence that it will prevent executable code to be extracted from the formalization. It would surely be desirable to be able to obtain certified algorithms for, in the present case, the simplification of context-free grammars. The alternative then would be to represent rules as a member of type \texttt{list (non\_terminal * sf)} instead. This, however, would have changed the whole declarative approach of the present work into a more computational one, by creating functions that manipulate grammars that have the desired properties. The purely logical approach, thus, was considered more appealing and selected as the choice for the present formalization. Anyway, it does not affect the objectives listed in Section 1 and can be adapted in the future in order to allow for code extraction, although this should demand a considerable effort in the creation and proof of program-related scripts.
The example below represents grammar

$$G = (\{S', A, B, a, b\}, \{a, b\}, \{S' \rightarrow aS', S' \rightarrow b\}, S')$$

that generates language $$a^*b$$:

### Inductive non_terminal

Type :=

| $$S'$$ |
| $$A$$ |
| $$B$$ |

### Inductive terminal

Type :=

| $$a$$ |
| $$b$$ |

### Inductive rs1

non_terminal $$\rightarrow$$ sf $$\rightarrow$$ Prop :=

| r1: $$\text{rs1} S' \rightarrow \text{inr } a; \text{inl } S'$$ |
| r2: $$\text{rs1} S' \rightarrow \text{inr } b$$ |

### Definition g1

start_symbol := $$S'$$;

rules := $$\text{rs1}$$;

rules_finite := $$\text{rs1_finite}$$. |

The term $$\text{rs1_finite}$$ (the proof that the set of rules of $$g_1$$ is finite) is not presented here, but can be easily constructed and is available from the link provided in Section 1.

Another fundamental concept used in this formalization is the idea of derivation: a grammar $$g$$ derives a string $$s_2$$ from a string $$s_1$$ if there exists a series of rules in $$g$$ that, when applied to $$s_1$$, eventually result in $$s_2$$. An inductive predicate definition of this concept in Coq ($$\text{derives}$$) uses two constructors.

### Inductive derives (g: cfg); sf $$\rightarrow$$ sf $$\rightarrow$$ Prop :=

| $$\text{derives_refl}$$: for all $$s$$: sf $$g$$,
| $$\text{derives} g s s$$ |
| $$\text{derives_step}$$: for all $$s_1$$ $$s_2$$ $$s_3$$: sf $$g$$,
| for all $$\text{left}$$: non_terminal $$g$$,
| for all $$\text{right}$$: sf $$g$$,
| $$\text{derives} g s_1 (s_2 ++ \text{inl } \text{left} :: s_3)$$ $$\rightarrow$$
| $$\text{rules} g \text{ left} \text{ right}$$ $$\rightarrow$$
| $$\text{derives} g s_1 (s_2 ++ \text{right} ++ s_3)$$ |

The constructors of this definition ($$\text{derives_refl}$$ and $$\text{derives_step}$$) are the axioms of our theory. Constructor $$\text{derives_refl}$$ asserts that every sentential form $$s$$ can be derived from $$s$$ itself. Constructor $$\text{derives_step}$$ states that if a sentential form that contains the left-hand side of a rule is derived by a grammar, then the grammar derives the sentential form with the left-hand side replaced by the right-hand side of the same rule. This case corresponds to the application of a rule in a direct derivation step.

A grammar generates a string if this string can be derived from its root symbol. Finally, a grammar produces a sentence if it can be generated from its root symbol.

### Definition generates (g: cfg) (s: sf): Prop :=

derives $$g$$ [inl (start_symbol $$g$$)] $$s$$.

### Definition produces (g: cfg) (s: sentence): Prop :=

generates $$g$$ (map terminal_lift $$s$$).

Function $$\text{terminal_lift}$$ converts a terminal symbol into an ordered pair of type (non_terminal + terminal). With these definitions, it has been possible to prove various lemmas about grammars and derivations, and also operations on grammars,
all of which were useful when proving the main theorems of this article.

As an example, the lemma that states that $G$ produces the string $aab$ (that is, $aab \in L(G)$) is represented as:

\[
\text{Lemma } G \text{ produces } aab:
\text{ produces } G [a; a; b].
\]

The proof of this lemma can be easily constructed and relates directly to the derivations in $S \Rightarrow aS \Rightarrow aaS \Rightarrow aab$, however in reverse order because of the way that derives is defined.

3 Simplification

The definition of a context-free grammar allows for the inclusion of symbols and rules that might not contribute to the language being generated. Also, context-free grammars might also contain sets of rules that can be substituted by equivalent smaller and simpler sets of rules. Unit rules, for example, do not expand sentential forms (instead, they just rename the symbols in them) and empty rules can cause them to contract. Although the appropriate use of these features can be important for human communication in some situations, this is not the general case, since it leads to grammars that have more symbols and rules than necessary, making difficult its comprehension and manipulation. Thus, simplification is an important operation on context-free grammars.

Let $G$ be a context-free grammar, $L(G)$ the language generated by this grammar and $\epsilon$ the empty string. Different authors use different terminology when presenting simplification results for context-free grammars. In what follows, we adopt the terminology and definitions of [33].

Context-free grammar simplification comprises four kinds of objects, whose definitions and results are presented below:

(i) An empty rule $r \in P$ is a rule whose right-hand side $\beta$ is empty (e.g. $X \rightarrow \epsilon$). We formalize that for all $G$, there exists $G'$ such that $L(G) = L(G')$ and $G'$ has no empty rules, except for a single rule $S \rightarrow \epsilon$ if $\epsilon \in L(G)$; in this case, $S$ (the initial symbol of $G'$) does not appear in the right-hand side of any rule in $G'$;

(ii) A unit rule $r \in P$ is a rule whose right-hand side $\beta$ contains a single non-terminal symbol (e.g. $X \rightarrow Y$). We formalize that for all $G$, there exists $G'$ such that $L(G) = L(G')$ and $G'$ has no unit rules;

(iii) $s \in V$ is useful ([33], p. 116) if it is possible to derive a string of terminal symbols from it using the rules of the grammar. Otherwise $s$ is called an useless symbol. A useful symbol $s$ is one such that $s \Rightarrow^* \omega$, with $\omega \in \Sigma^*$. Naturally, this definition concerns mainly non-terminals, as terminals are trivially useful. We formalize that, for all $G$ such that $L(G) \neq \emptyset$, there exists $G'$ such that $L(G) = L(G')$ and $G'$ has no useless symbols;

(iv) $s \in V$ is accessible ([33], p. 119) if it is part of at least one string generated from the root symbol of the grammar. Otherwise it is called an inaccessible symbol. An accessible symbol $s$ is one such that $S \Rightarrow^* \alpha s \beta$, with $\alpha, \beta \in V^*$. We formalize that for all $G$, there exists $G'$ such that $L(G) = L(G')$ and $G'$ has no inaccessible symbols.
Finally, we formalize a unification result: that for all $G$, if $G$ is non-empty, then there exists $G'$ such that $L(G) = L(G')$ and $G'$ has no empty rules (except for one, if $G$ generates the empty string), no unit rules, no useless symbols and no inaccessible symbols.

In all these four cases and five grammars that are discussed next (namely $g_{\text{emp}}$, $g_{\text{emp}}'$, $g_{\text{unit}}$, $g_{\text{use}}$ and $g_{\text{acc}}$), the proof of the predicate $\text{rules\_finite}$ is based on the proof of the correspondent predicate for the argument grammar. Thus, all new grammars satisfy the $\text{cfg}$ specification and are finite as well.

### 3.1 Empty rules

Result (i) is achieved in two steps. First, the idea of a nullable symbol was represented by the definition $\text{empty}$:

**Definition** $\text{empty}$

$\text{empty}(g: \text{cfg terminal _}) (s: \text{non-terminal + terminal}): \text{Prop} :=$

$\text{derivs} \ g \ [s] \ [\].$

Notation $sf'$ represents a sentential form built with symbols from $\text{non-terminal'}$ and $\text{terminal}$. Definition $\text{symbol\_lift}$ maps a pair of type $(\text{non-terminal + terminal})$ into a pair of type $(\text{non-terminal'} + \text{terminal})$ by replacing each $\text{non-terminal}$ with the corresponding $\text{non-terminal'}$:

**Inductive** $\text{non-terminal'}$: $\text{Type} :=$

$| \text{Lift\_nt}: \text{non-terminal} \rightarrow \text{non-terminal'}$

$| \text{New\_ss}.$

**Notation** $sf' := (\text{list} (\text{non-terminal'} + \text{terminal})).$

**Definition** $\text{symbol\_lift}$

$s: \text{non-terminal + terminal}): \text{non-terminal'} + \text{terminal} :=$

$\text{match} \ s \ \text{with} \ |
\text{inr} \ t \Rightarrow \text{inr} \ t$

$| \text{inl} \ n \Rightarrow \text{inl} (\text{Lift\_nt} \ n)$

end.

With these, a new grammar $g_{\text{emp}} \ g$ has been created, such that the language generated by it matches the language generated by the original grammar ($g$), except for the empty string. Predicate $g_{\text{emp\_rules}}$ states that every non-empty rule of $g$ is also a rule of $g_{\text{emp}} \ g$, and also adds new rules to $g_{\text{emp}} \ g$ where every possible combination of nullable non-terminal symbols that appears in the right-hand side of a rule of $g$ is removed, as long as the resulting right-hand side is not empty. Finally, it adds a rule that maps a new symbol, the start symbol of the new grammar ($\text{New\_ss}$), to the start symbol of the original grammar. For this reason, the new type $\text{non-terminal'}$ has been defined. The motivation for introducing a new start symbol at this point is to be able to prove that the start symbol does not appear in the right-hand side of any rule of the new grammar, a result that will be important in future developments.

**Inductive** $g_{\text{emp\_rules}} (g: \text{cfg _ }): \text{non-terminal } \rightarrow sf' \rightarrow \text{Prop} :=$

$| \text{Lift\_direct} :$

$\text{forall left: non-terminal,}$

$\text{forall right: sf,}$

$\text{right} \ [\] \rightarrow \text{rules} \ g \ \text{left right} \rightarrow$

$g_{\text{emp\_rules}} \ g \ (\text{Lift\_nt} \ \text{left}) \ (\text{map} \ \text{symbol\_lift} \ \text{right})$

$| \text{Lift\_indirect} :$

$\text{forall left: non-terminal,}$

$\text{forall right: sf,}$
Suppose, for example, that $S, A, B, C$ are non-terminals, of which $A, B$ and $C$ are nullable, $a, b$ and $c$ are terminals and $X \rightarrow aAbBcC$ is a rule of $g$. Then, the above definitions assert that $X \rightarrow aAbBcC$ is a rule of $g_{emp}$, and also:

- $X \rightarrow aAbBc$;
- $X \rightarrow abBcC$;
- $X \rightarrow aAbcC$;
- $X \rightarrow aAbc$;
- $X \rightarrow abBc$;
- $X \rightarrow abcC$;
- $X \rightarrow abc$.

Observe that grammar $g_{emp}$ does not generate the empty string. The second step, thus, was to define $g_{emp'}$, such that $g_{emp'}$ generates the empty string if $g$ generates the empty string. This was done by stating that every rule from $g_{emp}$ is also a rule of $g_{emp'}$ and also by adding a new rule that allow $g_{emp'}$ to generate the empty string directly if necessary.

Note that the generation of the empty string by $g_{emp'}$ depends on $g$ generating the empty string. The proof of the correctness of these definitions is achieved through the following theorem:

**Theorem** $g_{emp'}$ correct:

For all $g$ and $g_{emp'}$ such that $g_{equiv}(g_{emp'} g)$ $\land$ ($g_{generates_empty}$ $\Rightarrow$ has_one_empty_rule($g_{emp'} g$)) $\land$
Four auxiliary predicates have been used in this statement: \(g_{\text{equiv}}\) for two context-free grammars that generate the same language, \(g_{\text{generates empty}}\) for a grammar whose language includes the empty string, \(g_{\text{has one empty rule}}\) for a grammar that has an empty rule whose left-hand side is the initial symbol, and all other rules are not empty and \(g_{\text{has no empty rules}}\) for a grammar that has no empty rules at all.

The definition of \(g_{\text{equiv}}\) is straightforward:

\[
\text{Variables: } \text{non-terminal non-terminal terminal: Type.}
\]

\text{Definition } g_{\text{equiv}} (g_1: \text{cfg non-terminal terminal}) (g_2: \text{cfg non-terminal terminal}): \text{Prop} :=

forall s: \text{sentence},
produces g_1 s \leftrightarrow produces g_2 s.

When applied to the previous theorem, it translates into:

forall s: \text{sentence},
produces \((g_{\text{emp'}} g)\) s \leftrightarrow produces g s.

For the \(\rightarrow\) part, the strategy adopted is to prove that for every rule \(\text{left } \rightarrow g_{\text{emp'}} \text{right}\) of \((g_{\text{emp'}} g)\), either \(\text{left } \rightarrow g \text{right}\) is a rule of g or \(\text{left } \Rightarrow^* g \text{right}\) in g. For the \(<\) part, the strategy is a more complicated one, and involves induction over the number of derivation steps in g.

### 3.2 Unit rules

For result (ii), definition \(g_{\text{unit}}\) expresses the relation between any two non-terminal symbols \(X\) and \(Y\), and is true when \(X \Rightarrow^* Y\).

\text{Inductive } g_{\text{unit}} (g: \text{cfg terminal non-terminal}) (a: \text{non-terminal}): \text{non-terminal } \rightarrow \text{Prop} :=

| \text{unit_rule: } forall (b: \text{non-terminal}),
\text{rules g a } [\text{inl } b] \rightarrow \text{unit } g a b |
| \text{unit_trans: } forall b c: \text{non-terminal},
\text{unit } g a b \rightarrow 
\text{unit } g b c \rightarrow 
\text{unit } g a c |

Grammar \(g_{\text{unit}}\) represents the grammar whose unit rules have been substituted by equivalent ones. The idea is that \(g_{\text{unit}}\) g has all non-unit rules of g, plus new rules that are created by anticipating the possible application of unit rules in g, as informed by \(g_{\text{unit}}\).

\text{Inductive } g_{\text{unit rules}} (g: \text{cfg _ _ }): \text{non-terminal } \rightarrow \text{sf } \rightarrow \text{Prop} :=

| \text{Lift_direct': }:
\text{forall left: \text{non-terminal},
forall right: \text{sf},
(forall r: \text{non-terminal},
right \leftrightarrow [\text{inl } r]) \rightarrow \text{rules g left right } \rightarrow 
g_{\text{unit rules}} g \text{ left right} |
| \text{Lift_indirect': }:
\text{forall a b: \text{non-terminal},
unit g a b \rightarrow 
forall right: \text{sf},
\text{rules g b right } \rightarrow 
(forall c: \text{non-terminal},
right \leftrightarrow [\text{inl } c]) \rightarrow 
g_{\text{unit rules}} g a b |

\text{Definition } g_{\text{unit}} (g: \text{cfg _ _ }): \text{cfg _ _ } := \{|}
Finally, the correctness of $g_{\text{unit}}$ comes from the following theorem:

**Theorem** $g_{\text{unit correct}}$:

forall $g$: cfg _,
g_equiv ($g_{\text{unit}}$ $g$) $g$ /
has_no_unit_rules ($g_{\text{unit}}$ $g$).

The predicate has_no_unit_rules states that the argument grammar has no unit rules at all.

Similar to the previous case, for the $\rightarrow$ part of the $g_{\text{equiv}}$ ($g_{\text{unit}}$ $g$) $g$ proof, the strategy adopted is to prove that for every rule $\text{left} \rightarrow_{g_{\text{unit}}} \text{right}$ of ($g_{\text{unit}}$ $g$), either $\text{left} \rightarrow_{g} \text{right}$ is a rule of $g$ or $\text{left} \Rightarrow^* \text{right}$ in $g$. For the $\leftarrow$ part, the strategy is also a more complicated one, and involves induction over a predicate that is isomorphic to derives ($\text{derives3}$), but generates the sentence directly without considering the application of a sequence of rules, which allows one to abstract the application of unit rules in $g$.

### 3.3 Useless symbols

For result (iii), the idea of a useful symbol is captured by the definition useful.

**Definition** useful ($g$: cfg _) (s: non_terminal + terminal): Prop:=

match s with
| inst t => True
| inl n => exists s: sentence, derives g [inl n] (map term_lift s)
end.

The removal of useless symbols comprises, first, the identification of useless symbols in the grammar and, second, the elimination of the rules that use them. Definition $g_{\text{use rules}}$ selects, from the original grammar, only the rules that do not contain useless symbols. The new grammar, without useless symbols, can then be defined as in $g_{\text{use}}$.

**Inductive** $g_{\text{use rules}}$ ($g$: cfg): non_terminal $\rightarrow$ sf $\rightarrow$ Prop :=

| Lift_use : forall left: non_terminal,
|forall right: sf,
rules g left right $\rightarrow$
useful g (inl left) $\rightarrow$
(forall s: non_terminal + terminal, In s right $\rightarrow$
useful g s) $\rightarrow$ g_use_rules g left right.

**Definition** $g_{\text{use}}$ ($g$: cfg _): cfg _ := {};
start_symbol:= start_symbol _;
rules:= g_use_rules g;
rules_finite:= g_use_finite g |}.
\textbf{Theorem} \textit{g\_use\_correct}: \\
\textit{forall} \quad \textit{g}: \textit{cfg} \_ \_ , \\
\textit{non\_empty} \quad \textit{g} \rightarrow \\
\textit{g\_equiv} (\textit{g\_use} \quad \textit{g}) \quad \textit{g} /\backslash \\
\textit{has\_no\_useless\_symbols} (\textit{g\_use} \quad \textit{g}).

The predicates \textit{non\_empty}, and \textit{has\_no\_useless\_symbols} used above assert, respectively, that grammar \textit{g} generates a language that contains at least one string (which in turn may or may not be empty) and the grammar has no useless symbols at all.

The \textit{\rightarrow} part of the \textit{g\_equiv} proof is straightforward, since every rule of \textit{g\_use} is also a rule of \textit{g}. For the converse, it is necessary to show that every symbol used in the derivation of \textit{g} is useful, and thus the rules used in this derivation also appear in \textit{g\_use}.

3.4 Inaccessible symbols

Result (iv) is similar to the previous case, and definition \textit{accessible} has been used to represent accessible symbols in context-free grammars.

\textbf{Definition} \textit{accessible} \\
\textit{(g: cfg)} (\textit{s: non\_terminal} + \textit{terminal}): \textit{Prop} := \\
\textit{exists} \quad \textit{s1} \quad \textit{s2}: \textit{sf}, \textit{derives} \quad \textit{g} \quad (\textit{inl} (\textit{start\_symbol} \quad \textit{g})) \quad (\textit{s1++s::s2}).

Definition \textit{g\_acc\_rules} selects, from the original grammar, only the rules that do not contain inaccessible symbols. Definition \textit{g\_acc} represents a grammar whose inaccessible symbols have been removed.

\textbf{Inductive} \textit{g\_acc\_rules} \textit{(g: cfg)}: \textit{non\_terminal} \rightarrow \textit{sf} \rightarrow \textit{Prop} := \\
| \textit{Lift\_acc} : \textit{forall} \quad \textit{left}: \textit{non\_terminal}, \\
\textit{forall} \quad \textit{right}: \textit{sf}, \\
\textit{rules} \quad \textit{g} \quad \textit{left} \quad \textit{right} \rightarrow \textit{accessible} \quad \textit{g} \quad (\textit{inl} \quad \textit{left}) \rightarrow \\
\textit{g\_acc\_rules} \quad \textit{g} \quad \textit{left} \quad \textit{right}.

\textbf{Definition} \textit{g\_acc} \textit{(g: cfg} \_ \_): \textit{cfg} \_ \_ := \{ \\
\textit{start\_symbol}:= \textit{start\_symbol} \quad \textit{g}; \\
\textit{rules}:= \textit{g\_acc\_rules} \quad \textit{g}; \\
\textit{rules\_finite}:= \textit{g\_acc\_finite} \quad \textit{g} \ \}.

The correctness of the inaccessible symbol elimination operation can be certified by proving theorem \textit{g\_acc\_correct}, which states that every context-free grammar generates a language that can also be generated by an equivalent context-free grammar whose symbols are all accessible.

\textbf{Theorem} \textit{g\_acc\_correct}: \\
\textit{forall} \quad \textit{g}: \textit{cfg} \_ \_ , \\
\textit{g\_equiv} (\textit{g\_acc} \quad \textit{g}) \quad \textit{g} /\backslash \\
\textit{has\_no\_inaccessible\_symbols} (\textit{g\_acc} \quad \textit{g}).

In a way similar to \textit{has\_no\_useless\_symbols}, the absence of inaccessible symbols in a grammar is expressed by predicate \textit{has\_no\_inaccessible\_symbols} used above.

Similar to he previous case, the \textit{\rightarrow} part of the \textit{g\_equiv} proof is also straightforward, since every rule of \textit{g\_acc} is also a rule of \textit{g}. For the converse, it is necessary to show that every symbol used in the derivation of \textit{g} is accessible, and thus the rules used in this derivation also appear in \textit{g\_acc}. 176
If one wants to obtain a new grammar simultaneously free of empty and unit rules, and of useless and inaccessible symbols, it is not enough to consider the previous independent results. On the other hand, it is necessary to establish a suitable order to apply these simplifications, in order to guarantee that the final result satisfies all desired conditions. Then, it is necessary to prove that the claims do hold.

For the order, we should start with (i) the elimination of empty rules, followed by (ii) the elimination of unit rules. The reason for this is that (i) might introduce new unit rules in the grammar, and (ii) will surely not introduce empty rules, as long as original grammar is free of them (except for $S \rightarrow \epsilon$, in which case $S$, the initial symbol of the grammar, must not appear in the right-hand side of any rule). Then, elimination of useless and inaccessible symbols (in either order) is the right thing to do, since they only remove rules from the original grammar (which is specially important because they do not introduce new empty or unit rules).

The formalization of this result is captured in the following theorem, which represents the main result of this work:

\[
\text{Theorem } g\text{-simpl-exists-v1:}
\]

\[
\forall g : \text{cfg non-terminal terminal},
\]

\[
\text{non-empty } g \rightarrow
g \exists g' : \text{cfg (non-terminal' non-terminal) terminal},
\]

\[
g\text{-equiv } g' \land
\]

\[
\text{has-no-inaccessible-symbols } g' \land
\]

\[
\text{has-no-useless-symbols } g' \land
\]

\[
\text{generates-empty } g' \rightarrow \text{has-one-empty-rule } g' \land
\]

\[
\text{has-no-empty-rules } g' \land
\]

\[
\text{start-symbol-not-in-rhs } g'.
\]

Hypothesis non-empty $g$ is necessary in order to allow the elimination of useless symbols. The predicate start_symbol_not_in_rhs states that the start symbol does not appear in the right-hand side of any rule of the argument grammar.

The proof of $g\text{-simpl-exists-v1}$ demands auxiliary lemmas to prove that the characteristics of the initial transformations are preserved by the following ones. For example, that all of the unit rules elimination, useless symbol elimination and inaccessible symbol elimination operations preserve the characteristics of the empty rules elimination operation.

The proofs of all lemmas and theorems presented in this article have been formalized in Coq and comprises approximately 10,000 lines of scripts. This number can be explained for the following reasons:

(i) The style adopted for writing the scripts: for the sake of clarity, each tactic is placed in its own line, despite the possibility of combining several tactics in the same line. Also, bullets (for structuring the code) were used as much as possible and the sequence tactical (using the semicolon symbol) was avoided at all. This duplicates parts of the code but has the advantage of keeping the static structure of the script related to its dynamic behaviour, which favors legibility and maintenance.

(ii) The formalization includes not only the main theorems described here, but also an extensive library of other fundamental and auxiliary lemmas on context-free grammars and derivations, which have been used to obtain the main results.
presented here, were used in the previously obtained results and will be used in future developments.

4 Further Work

Current work has focussed on the representation of context-free grammars, context-free derivations, the formalization of grammar simplification strategies and the certification of their correctness. It represents an important step towards the formalization of context-free language theory, and adds to the previous results on the formalization of closure properties for context-free grammars ([30]).

The next steps of this formalization work are:

(i) Describe Chomsky normal form for context-free grammars and prove its existence for any context-free grammar that satisfies the required conditions;

(ii) Obtain a formal proof of the decidability of the membership problem for context-free languages;

(iii) Obtain a formal proof of the Pumping Lemma for context-free languages.

The second and the third objectives will rely on the first one.

5 Related Work

Language and automata theory has been subject of formalization since the mid-1980s, when Kreitz used the Nuprl proof assistant to prove results about deterministic finite automata and the pumping lemma for regular languages [25]. Since then, the theory of regular languages has been formalized partially by different researchers using different proof assistants (see [11], [22], [16], [10], [26], [27], [2], [1], [28] [8], [9], [3], [13], [24] and [34]). The most recent and complete formalization, however, is the work by Jan-Oliver Kaiser [14], which used Coq and the SSReflect extension to prove the main results of regular language theory.

Context-free language theory has not been formalized to the same extent so far, and the results were obtained with a diversity of proof assistants, including Coq, HOL4 and Agda. Most of the effort start in 2010 and has been devoted to the certification and validation of parser generators. Examples of this are the works of Koprowski and Binsztok (using Coq, [23]), Ridge (using HOL4, [32]), Jourdan, Pottier and Leroy (using Coq, [21]) and, more recently, Firsov and Uustalu (in Coq, [17]).

On the more theoretical side, on which the present work should be considered, Norrish and Barthwal (using HOL4, [4], [5], [6]), published on general context-free language theory formalization, including the existence of normal forms for grammars, pushdown automata and closure properties. Recently, Firsov and Uustalu proved the existence of a Chomsky Normal Form grammar for every general context-free grammar (using Agda, [18]).

It can thus be noted that apparently no formalization has been done in Coq so far for results not related directly to parsing and parser verification, and that this constitutes an important motivation for the present work, mainly due to the increasing usage and importance of Coq in different areas and communities. Specifically,
the formalization done by Norrish and Barthwal in HOL4 is quite comprehensive and extends our work with the Greibach Normal Form and pushdown automata and its relation to context-free grammars. It does not include, however, a proof of either the decidability of the membership problem or the Pumping Lemma for context-free languages, which are objectives of the present work. The formalization by Firsov and Uustalu in Agda comprises basically the existence of a Chomsky Normal Form, and formalizes the elimination of empty and unit rules, but not elimination of useless and inaccessible symbols.

When it comes to computability theory and Turing machines related classes of languages, formalization has been approached by Asperti and Ricciotti (Matita, [3]), Xu, Zhang and Urban (Isabelle/HOL, [35]) and Norrish (HOL4, [29]).

6 Conclusions

The present paper reports an ongoing effort towards formalizing the classical context-free language theory, initially based only on context-free grammars, in the Coq proof assistant. All important objects have been formalized and different simplification strategies on grammars have been implemented. Proofs of their correctness were successfully constructed.

Building up on the previous formalization of closure properties for context-free grammars [30], the present results create a comfortable situation in order to pursue the formalization of normal forms for context-free grammars, the next step of this work.

The authors acknowledge the fruitful discussions and contributions of Nelma Moreira (Departamento de Ciência de Computadores da Faculdade de Ciências da Universidade do Porto, Portugal) and José Carlos Bacelar Almeida (Departamento de Informática da Universidade do Minho, Portugal) to this work. Also, we are grateful to the anonymous reviewers who provided useful criticisms and insights, and contributed to a better presentation of this work.

References


180


Nondeterministic Linear Automata and a Class of Deterministic Linear Languages

Benjamín Callejas Bedregal
Departamento de Informática e Matemática Aplicada – DIMAp
Universidade Federal do Rio Grande do Norte – UFRN
Natal, Brazil

Abstract
In this paper we consider the class of \(\lambda\)-nondeterministic linear automata as a model of the class of linear languages. As usual in other automata models, \(\lambda\)-moves do not increase the acceptance power. The main contribution of this paper is to introduce the deterministic linear automata and even linear automata, i.e. the natural restriction of nondeterministic linear automata for the deterministic and even linear language classes. In particular, there are different not equivalent proposed for the class of “deterministic” linear languages and here we proved that the class of languages accepted by the proposed deterministic linear automata are not contained in any of the these classes and in fact contain properly the most of these classes.

Keywords: Linear languages, nondeterministic linear automata, deterministic linear automata, even linear languages, \(\lambda\)-moves, minimization of states.

1 Introduction

The class of linear languages, also known by linear context-free languages, is situated in the extended Chomsky hierarchy between the classes of regular and context-free languages. Linear grammars (the more well know model for the class of linear languages), allows have a controle on matches between leftmost symbols with rightmost symbols of a substring. This capability, makes that the most commonly used examples of context-free languages are in that class. Thus, for example, the classical context-free languages (palindromes and \(a^nb^n\)) which often are used as examples of non-regular languages are linear languages. An example of a context-free languages which is not linear is \(\{a^nb^ma^nb^n : n, m \geq 1\}\) [3].

The most usual model for linear languages are the linear grammars and some normal form for them had been proposed. In term of automata counterpart, it is possible to find at least four different models: Two-tape nondeterministic finite automata of a special type [11,27], finite transducer [23,27], one-turn pushdown automata [3,9,11,14,15] and right-left-monotone restarting automata [18]. Although the merits of each model, they are not intrinsic, in the sense that their definition includes external elements. For example, in the two-tape nondeterministic finite

Selected, revised papers will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
automata of a special type the input is split in two tapes with the second one containing the reverse of the right part of the input string. Notice that, all the usual type of automata – e.g. finite automata, pushdown automata (PDA in short), Turing machines, etc. – assume that the input string is sequentially dispossed, without any modification, in the input tape at the start of execution that here we call of natural condition. Thus, this model does not satisfy the natural condition and is not intrinsic because the choice of how and where to divide the input and the application of the reversion is external to the model.

Analogously, the finite transducer model also presuppose that externally the input string is divided in two parts which are separated by a special symbol and where the second part is reverted. Therefore, this model also presuppose a previous knowledge of where must be divided the input. Thus, this model also not satisfy the natural condition and consider external agents. On the other hand, a turn in a PDA is a move which decreases the stack and which is preceded by a move that increase the stack, the one-turn PDA model (a PDA which makes at most one turn in the stack). Determine the maximal possible quantity of turns that a PDA could make (i.e. consider all possible input string) require an external control. Thus, one-turn PDA model is not intrinsic. Finally, the right-left-monotone restarting automata introduced in [18] is a restriction of a more general class of automata, the restarting automata introduced in [16,17]. Since in this automata the transitions work with strings instead of symbols and allow actions such as the reversion of strings, this model of linear language is of higher level and more complex than the other models. On the other hand, this model requires that at each cycle the distance from the actual place where a rewrite takes place to the right and left end of the tape must not increase, however the verification of that condition can not be made internally in the model and therefore also is a non intrinsic model.

In this paper we consider the automata model for linear languages introduced in [5], called of λ-nondeterministic linear automata, which extend the notion of nondeterministic finite automata with λ-moves, λ-NFA in short, and therefore contain properly this class of automata. This automata model, in our viewpoint, not consider external elements, is not a subclass of automata used to model a more broad class of languages, is strongly inspired in a formal norm introduced here for linear grammars and therefore is simpler than the other models. In addition, the λ-moves in the usual classes of automata are not essentials for the models. Thus, for example, λ-moves do not increase the acceptance power of finite automata (see, for example, [15]), of pushdown automata (see, for example, [11]) and of Turing machines (by Church-thesis). Analogously, in [5] was proved that λ-moves also do not increase the acceptance power of λ-nondeterministic linear automata.

In this work we introduce the deterministic version of this automata, called deterministic linear automata, DLA in short, and compare the class of languages accepted by DLA’s with the class DL introduced by de la Higuera and Oncina in [13] and with the class of linear languages and deterministic context-free languages. We also provide a characterization of the NLA which are equivalent to some DLA. In this paper also we determine an infinite hierarchy of class of formal languages which are in the class of languages accepted by DLA’s and the class of linear languages. Finally, we show how the class of even linear languages, introduced originally by
Amar and Putzolu in [1], can be captured in the nondeterministic linear automata model.

2 Linear grammars

As usual a formal grammar $G$ is a tuple $(V, T, S, P)$ where $V$ is a finite set of variables, $T$ is a set of terminal symbols and therefore $V \cap T = \emptyset$, $S \in V$ is the start variable and $P \subseteq (V \cup T)^{+} \times (V \cup T)^{*}$ is the set of productions. Ordered pairs $(x, y) \in P$ are denoted by $x \rightarrow y$.

A grammar $G = (V, T, S, P)$ is linear, if each production in $P$ has a variable in their left side and has at most one variable in their right side, without restriction on the position of this variable. A variable $A \in V$ will be called left linear if in each production $A \rightarrow y \in P$, either $y \in T^{*}$ or $y = Bz$ for some $z \in T^{*}$ and $B \in V$. Analogously, a variable $A$ will be called right linear if in each production $A \rightarrow y \in P$, either $y \in T^{*}$ or $y = zB$ for some $z \in T^{*}$ and $B \in V$. A linear grammar $G$ is in linear normal form, in short lnf, if each variable $A \in V$ is left or right linear. Thus a variable $A$ in a linear grammar is both, left and right linear, just when each production having $A$ in the left side has at the right side either a string of terminal symbols (inclusive the empty string) or a single variable.

Notice that our linear normal form is subtly different from the usual linear normal form (e.g. see [11,21,28]) where is tolerate two productions with the same variable in the leftmost and rightmost position, respectively.

**Example 1** The grammar $G = (\{S\}, \{a, b\}, S, P)$ where $P$ is given by

\[
S \rightarrow aSb \mid aSbb \mid aSbbb \mid ab \mid abb
\]

is linear but is not in the lnf.

As usual, for each $u, v, w \in (V \cup T)^{*}$ and $A \in V$, $uAw \Rightarrow uvw$ if there is a production $A \rightarrow v \in P$. Let $\Rightarrow^{*}$ be the reflexive and transitive closure of $\Rightarrow$. The language generated by a linear grammar $G$ is

\[
\mathcal{L}(G) = \{w \in T^{*} : S \Rightarrow^{*} w\}
\]

In the Example 1,

\[
\mathcal{L}(G) = \{a^{m}b^{n} : 1 \leq m \leq n \leq 3m\}
\]

Languages generates by linear grammars are called linear languages.

**Lemma 1** [5] Let $G$ be a linear grammar. Then there exists a linear grammar $G'$ in the lnf such that $\mathcal{L}(G) = \mathcal{L}(G')$.

**Example 2** The lnf of $G$ in Example 1 obtained following the algorithm given in the proof of the Lemma 1 given in [5, Lemma 2.1] is the grammar $G' = (\{S, A\}, \{a, b\}, S, P')$ where $P'$ is given by

\[
S \rightarrow aA \mid ab \mid abb \mid abbb
\]

$A \rightarrow Sb \mid Sbb \mid Sbbb$

185
A linear grammar $G$ is in **strong linear normal form**, in short slnf, if it is in lnf and the right side of each production is of the form $aA$ or $Aa$ where $a \in T \cup \{\lambda\}$ and $A \in V \cup \{\lambda\}$.

**Proposition 2** [5] Let $G$ be a linear grammar. Then there exists a linear grammar $\hat{G}$ in slnf such that $L(G) = L(\hat{G})$.

**Example 3** The slnf of $G'$ in Example 2 obtained following the algorithm given in the proof of the Proposition 2 given in [5, Prop. 2.1.] is the grammar $\hat{G} = \langle \{S,A,B,C,D,E,F\}, \{a,b\}, S, \hat{P} \rangle$ where $\hat{P}$ is given by

\[
\begin{align*}
S & \rightarrow aA \mid aD \\
A & \rightarrow Sb \mid Bb \\
B & \rightarrow Sb \mid Cb \\
C & \rightarrow Sb \\
D & \rightarrow b \mid bE \\
E & \rightarrow b \mid bF \\
F & \rightarrow b
\end{align*}
\]

### 3 $\lambda$-Nondeterministic linear automata

A **$\lambda$-nondeterministic linear automata**, $\lambda$-nla in short, consist of two disjoint finite sets of states ($Q_L$ and $Q_R$) some of which will be consider as accepting states, an input tape which is divided in cell and is not limited at the right, each cell can hold a symbol from a finite input alphabet, two read heads and a control unit which manage the behavior of the $\lambda$-nla in accord with the current configuration. The execution of a $\lambda$-nla start with a string in the input tape, with the left read head pointing to leftmost symbols, the right read head pointing to the rightmost symbol and the current state being a state of an special set of states of $Q_L \cup Q_R$ called set of start states\(^1\). A computation step in a $\lambda$-nla is made as follow: the control unit depending of the class which belong the current state (in the first time $Q_L$) uses the left or the right read head to scan a symbol from the tape, move the left read head one cell to the right if the current state is in $Q_L$ or move the right read head one cell to the left if the current state is in $Q_R$, and making a nondeterministic choice, change the state choosing it from a set of possible states. The control unit of a $\lambda$-nla also allows changing of state without move the read heads, i.e. without read an input symbol. The computations halts when a read head pass over the other read head or when there is no choice of possible of action. A string is only accepted when the automata halts in a final state or one read head pass over the other read head.

The Figure 1 illustrate an schematic representation of a $\lambda$-nla.

Formally, a $\lambda$-nla is a sextuple $M = \langle Q_L, Q_R, \Sigma, \delta, I, F \rangle$ where $Q_L$ and $Q_R$ are

---

\(^1\) The use of a set of start states, although of no usual, had been used in several models of automata. For example, [7, Def.4.1] and [19, page 32] in non-deterministic finite automata and [8, page 89], [11, page 52] in transition systems.
disjoint and finite sets of states, $\Sigma$ is a finite set of input symbols (the alphabet), $I \subseteq Q_L \cup Q_R$ is the set of start states, $F \subseteq (Q_L \cup Q_R)$ is the set of final or accepting states and $\delta : (Q_L \cup Q_R) \times (\Sigma \cup \{\lambda\}) \rightarrow \wp(Q_L \cup Q_R)$

Analogously to finite automata, each $\lambda$-NLA has associated a directed graph, called transition diagram. In order to distinguish states in $Q_L$ of states in $Q_R$ we use circles and squares to represent them.”

Notice that, if $M$ is a $\lambda$-NLA with $Q_R = \emptyset$, then $L(M)$ is a regular language. In fact, $M' = \langle Q_L, \Sigma, I, \delta, F \rangle$ is a $\lambda$-nondeterministic finite automaton (with a set of start states) such that $L(M') = L(M)$ and whose transition diagram is exactly the same than the transition diagram for $M$. Thus, $\lambda$-NLA is a natural extension of $\lambda$-nondeterministic finite automata.

**Example 4**  The Figure 2 illustrate the $\lambda$-NLA $M = \langle Q_L, Q_R, \Sigma, \delta, \{q_0\}, F \rangle$ where

- $Q_L = \{q_0, q_1, q_2, q_3\}$
- $Q_R = \{p_1, p_2, p_3, p_4\}$
- $\Sigma = \{a, b\}$
- $F = \{p_1, q_2\}$
- $\delta(q_0, a) = \{q_0, p_1\}$, $\delta(q_0, \lambda) = \{p_3\}$, $\delta(p_1, a) = \{p_2\}$, $\delta(p_2, a) = \{q_1\}$, $\delta(q_1, b) = \{p_1\}$, $\delta(p_3, b) = \{p_3, q_2\}$, $\delta(q_2, b) = \{q_3\}$, $\delta(q_3, b) = \{p_4\}$, $\delta(p_4, a) = \{q_2\}$ and empty for the remain (i.e. $\delta(q_0, b) = \emptyset$, $\delta(p_1, b) = \emptyset$, etc.).

An instantaneous description (ID) of a $\lambda$-NLA must record the current state, the string remain to read and which read head is active. Thus an ID is a pair $(q, w)$ in $(Q_L \cup Q_R) \times \Sigma^*$ meaning that remain to read the string $w$, the current state is $q$ and the read head which is active is the left when $q \in Q_L$ and is the right when $q \in Q_R$.

The symbol $\vdash_M$ denote a valid move from an ID to another ID in a $\lambda$-NLA $M$. When the subscript $M$ is clear we omit it. Thus, for each $q \in Q_L$, $q' \in Q_L \cup Q_R$, $p \in Q_R$, $w \in \Sigma^*$ and $a \in \Sigma \cup \{\lambda\}$

$(q, aw) \vdash (q', w)$ is possible if and only if $q' \in \delta(q, a)$ and $(p, wa) \vdash (q', w)$ is possible if and only $q' \in \delta(p, a)$.

We use $\vdash^*$ for the reflexive and transitive closure of $\vdash$, i.e. $\vdash^*$ represent moves involving an arbitrary number of steps. Thus, in the example 4, we have
that \((q_0, abbaaaa) \vdash (p_1, baa)\), because
\[
(q_0, abbaaaa) \vdash (p_1, bbaaaa) \vdash (q_1, bbaa) \vdash (p_1, baa)
\]
The language accepted by a \(\lambda\)-NLA \(M\) is the set
\[
L(M) = \{w \in \Sigma^* : (q_0, w) \vdash (q_f, \lambda) \text{ for some } q_0 \in I \text{ and } q_f \in F\}
\]
For the case of \(\lambda\)-NLA \(M\) in the Example 4,
\[
L(M) = \{a^mb^n : m \geq 1 \text{ and } n \geq 0\} \cup \{a^kb^2m \in \Sigma^* : k, m \geq 0 \text{ and } n \geq 1\}.
\]
Notice that, the halting mechanism of the NLA, in spite that is not explicit in their mathematical formulation, can be formalized by the ID notion as follow: when an ID \((q, \lambda)\) is achieve through a move (e.g. \((q', a) \vdash (q, \lambda))\), we are in the situation of “a read head pass over the other read head”.

3.1 \(\lambda\)-NLA and linear languages

First, we will prove that each language accepted by some \(\lambda\)-NLA \(M\) is linear, i.e.

**Theorem 3** [5] Let \(M = \langle Q_L, Q_R, \Sigma, I, F \rangle\) be a \(\lambda\)-NLA. Then exist a linear grammar \(G\) such that \(L(M) = L(G)\).

Conversely, each language generated by a linear grammar \(G\) is accepted by some \(\lambda\)-NLA.

**Theorem 4** [5] Let \(G = \langle V, T, S, P \rangle\) be a linear grammar. Then exist a \(\lambda\)-NLA \(M\) such that \(L(M) = L(G)\).

The algorithms in the proves (which can be found in [5]) of the Theorems 3 and 4 are duals, in the sense that applying one and next the other, we obtain the same object. Obviously, for that in the case of Theorem 4 the grammar must be in the SLNF.
3.2 \( \lambda \)-moves are not necessary

A \( \lambda \)-nla without \( \lambda \)-transitions is called \textbf{non-deterministic linear automaton}, in short \textbf{nla}.

The \( \lambda \)-moves in the usual classes of automata are not essential for the model. Thus, for example, \( \lambda \)-moves do not increase the computational power acceptance of finite automata (see, for example, [15]), of pushdown automata (see, for example, [11]) and of Turing machines (by Church-thesis). So its reasonable to hope that nla and \( \lambda \)-nla have the same acceptance power. Nevertheless, when a \( \lambda \)-transition in a \( \lambda \)-nla happens between two states of different type it is not obvious how we can eliminate it without changing the language accepted by the automaton.

**Theorem 5** [5] Let \( M = \langle Q_L, Q_R, \Sigma, \delta, I, F \rangle \) be a \( \lambda \)-nla. Then exist a nla \( M' \) such that \( \mathcal{L}(M) = \mathcal{L}(M') \).

4 Deterministic linear automata

There are several nonequivalent notions for “deterministic” linear languages (see for example [13]). The most general among those classes is \textbf{DL}, where a language \( L \in \text{DL} \) if it can be generated by a deterministic linear grammar, i.e. by a linear grammar \( G = (V, T, S, P) \) where each production has the form \( A \to aBu \) or \( A \to \lambda \) and if \( A \to aBu \), \( A \to aCv \in P \) then \( B = C \) and \( u = v \), with \( A, B, C \in V \) and \( u, v \in T^* \). As proved in [13], \textbf{DL} is a proper subset of \textbf{DCFL} \( \cap \text{Lin} \), where \textbf{DCFL} is the class of deterministic context-free languages and \textbf{Lin} the class of linear languages.

On the other hand, a nla is \textbf{deterministic}, \textbf{dla} in short, if for each \( a \in \Sigma \) and \( q \in (Q_L \cup Q_R) \), \( |\delta(q, a)| \leq 1 \). A linear language is deterministic if it is accepted by a dla. Thus, in an dla \( \delta(q, a) = \{q'\} \) or \( \delta(q, a) = \emptyset \). For that reason we consider the transition function of a dla \( M = \langle Q_L, Q_R, \Sigma, \delta, I, F \rangle \) as a partial function from \((Q_L \cup Q_R) \times \Sigma \) into \((Q_L \cup Q_R) \).

**Theorem 6** If \( L \in \text{DL} \) then there is a dla \( M \) such that \( \mathcal{L}(M) = L \).

**Proof.** (sketch) Let \( G \) be a deterministic linear grammar such that \( \mathcal{L}(G) = L \). Let \( G' \) be the slnf of \( G \) obtained following the algorithm in Proposition 2. By construction, if \( A \to aB \) and \( A \to aC \) then, \( B = C \), and if \( A \to Ba \) and \( A \to Ca \) then \( B = C \). Thus, \( M = \langle Q_L, Q_R, T, \delta, \{S\}, F \rangle \) where \( Q_L = \{ A \in V' : A \text{ is a right linear variable} \} \), \( Q_R = \{ A \in V' : A \text{ is a left linear variable} \} \), \( F = \{ A : A \to \lambda \} \) and \( \delta(A, a) = B \) whenever \( A \to aB \). Clearly, \( M \) is a dla.

**Corollary 7** Let \textbf{DLin} be the class of languages accepted by some dla. Then

\[
\text{DL} \subseteq \text{DLin}
\]

**Proof.** Straightforward from Theorem 6, \( \text{DL} \subseteq \text{DLin} \).

On the other hand, de la Higuera and Oncina in [13] state that the language \( \{a^n b^n : n \geq 1\} \cup \{a^n c^n : n \geq 1\} \notin \text{DL} \). Nevertheless the Figure 3 present a dla which accepts this language and therefore \( \text{DL} \subseteq \text{DLin} \).
In the following proposition we relate the class $\text{DLin}$ with the class of deterministic context-free languages, $\text{DCFL}$ in short.

**Proposition 8**

$$\text{DLin} - \text{DCFL} \neq \emptyset \text{ and } \text{DCFL} - \text{DLin} \neq \emptyset$$

**Proof.** The DLA in Figure 4 clearly accepts the language of palindromes over the alphabet $\{a, b\}$. But, this language is not in DCFL (see [15,20]). So, $\text{DLin} - \text{DCFL} \neq \emptyset$. Conversely, as is well known, the language $L = \{a^m b^n a^m b^n : m \geq 1 \text{ and } n \geq 1\} \in \text{DCFL} - \text{Lin}$ and therefore $\text{DCFL} - \text{DLin} \neq \emptyset$.

Each deterministic linear language is linear, but as it will be proved below, the converse does not hold.

**Proposition 9** Let $\text{Lin}$ be the class of linear languages. Then

$$\text{DLin} \subset \text{Lin}$$

**Proof.** (Sketch) Let $L(3)$ be the linear language in the equation (1). Suppose that $L(3) \in \text{DLin}$. Then, there is a DLA $M$ such that $\mathcal{L}(M) = L(3)$. Let $a^m b^n$ be a string in $L(3)$. Let $q_0 \in I$. We have two cases:
Case $q_0 \in Q_L$, then first $M$ must read the leftmost $a$ in $a^n b^n$. If $\delta(q_0, a) \in Q_R$ then the $a$ needs to match with one, two or three $b$'s what clearly requires a nondeterministic choice and therefore is a contradiction. If $\delta(q_0, a) = q \in Q_L$ then $M$ must read a new $a$ and again either $\delta(q, a) \in Q_R$ in which case $M$ must read one, two or three $b$'s or $\delta(q, a) \in Q_L$ in which case $M$ must read a new $a$, and so on. But in some moment $M$ should make a match between an $a$ with one, two or three $b$'s.

The case of $q_0 \in Q_R$ is analogous. Therefore, $L(3) \in \text{Lin} - \text{DLin}$. 

\[\square\]

As corollary of the Prop. 9, we have that there are \text{nla} for which there is no equivalent \text{dla}, i.e. a \text{dla} that accept the same linear language. However, the next theorem provide a characterization of the \text{nla} that are equivalent to some \text{dla}, i.e. \text{nla} which accept deterministic linear languages.

Given an \text{nla} $M_N = (Q_L, Q_R, \Sigma, \delta, I, F)$ define $\phi(Q)$ as the least set containing \{ \{q\} : q \in I\} and such that for each $X \in \phi(Q)$ and $a \in \Sigma$ we have that $\bigcup_{q \in X} \delta(q, a) \in \phi(Q)$. Thus, $\phi(Q) \subseteq \mathcal{P}(Q_L \cup Q_R)$.

**Theorem 10** Let $L \subseteq \Sigma^*$ be a language. Then $L \in \text{DLin}$ iff exists an \text{nla} $M_N = (Q_L, Q_R, \Sigma, \delta, I, F)$ such that $L(M_N) = L$ and for each $X \in \phi(Q)$, $X \subseteq Q_L$ or $X \subseteq Q_R$.

**Proof.** ($\Rightarrow$) If $L \in \text{DLin}$ then there is a \text{dla} $M_D = (Q_L, Q_R, \Sigma, \delta_D, I, F)$ such that $L(M_D) = L$. Let $M_N = (Q_L, Q_R, \Sigma, \delta, I, F)$ where $\delta(q, a) = \{\delta_D(q, a)\}$. Clearly, $M_N$ is an \text{nla} such that $L(M_N) = L$ and $\phi(Q) = \{\{q\} : q \in Q\}$. Therefore, for each $X \in \phi(Q)$, $X = \{q\}$ for some $q \in Q_L \cup Q_R$ and so, trivially, $X \subseteq Q_L$ or $X \subseteq Q_R$.

($\Leftarrow$) Let $M_D = (\phi(Q)_L, \phi(Q)_R, \Sigma, \delta_D, I', F')$ where $\phi(Q)_L = \{X \in \phi(W) : X \subseteq Q_L\}$, $\phi(Q)_R = \{X \in \phi(W) : X \subseteq Q_R\}$, $I' = \{\{q\} : q \in I\}$, $F' = \{X \in \phi(Q) : X \cap F \neq \emptyset\}$ and $\delta_D(X, a) = \bigcup_{q \in X} \delta(q, a)$ for each $a \in \Sigma$ and $X \in \phi(Q)$. Clearly, $M_D$ is \text{dla}. Moreover, if $w \in L(M_N)$, then $(q, w) \equiv_{M_N} (q_f, \lambda)$ for some $q_i \in I$ and $q_f \in F$. Therefore, if $w = a_1 \ldots a_n$ then there is states (possibly with repetitions) $q(1), \ldots, q(n) \in Q$ such that $q(0) = q_i$, $q(n) = q_f$ and $(q(0), a_1 \ldots a_n) \vdash_{M_N} (q(1), a_2 \ldots a_n) \vdash_{M_N} \ldots \vdash_{M_N} (q(n-1), a_n) \vdash_{M_N} (q(n), \lambda)$. An in this case, clearly, we have that $(q(0), a_1 \ldots a_n) \vdash_{M_D} (q(1), a_2 \ldots a_n) \vdash_{M_D} \ldots \vdash_{M_D} (q(n-1), a_n) \vdash_{M_D} (q(n), \lambda)$ when $q(j) = \{q(j)\}$ for each $j = 0, \ldots, n$. Therefore, $w \in L(M_D)$. \[\square\]

### 5 A countable hierarchy of linear languages

Pushdown automata, \text{pda} in short, has a similar characteristic to linear automata: their nondeterministic version is more powerful than their deterministic version. In \text{pda} this difference allowed define several way to measure nondeterminism and consequently determine several hierarchies of classes of context-free languages varying from \text{DCFL} into (in the limit) \text{CFL}, the class of Context-Free Languages
From that hierarchy of classes we can establish a countable hierarchy of classes of linear languages in a simple way. Consider the hierarchy $CFL(1), CFL(2), \ldots$ determined in [4], in this hierarchy $CFL(1) = DCFL$, $CFL(k) \subset CFL(k + 1)$ for each $k \geq 1$ and $\lim_{k \to \infty} CFL(k) = CFL$. So, defining $LL(k) = CFL(k) \cap Lin$ we will have a countable hierarchy $LL(1) \subset LL(2) \subset \ldots$ such that, $LL(1) = DCFL \cap Lin$ and the limit of the hierarchy is the class of linear languages, i.e. $\lim_{k \to \infty} LL(k) = Lin$. In the following we will provide a countable hierarchy for classes of linear languages starting from $DLin$ going, in the limits, to $Lin$.

**Definition 5.1** Let $M = (Q_L, Q_R, \Sigma, \delta, I, F)$ be an nla. The explicit nondeterministic degree of $M$ is the following:

$$Ndeg(M) = \left( \sum_{q \in Q_L \cup Q_R, a \in \Sigma} | \delta(q, a) | \right) - | \{ (q, a) \in Q \times \Sigma : \delta(q, a) \neq \emptyset \} |$$

Let $Lin(k)$ the class of linear languages which can be accepted by an nla with explicit nondeterministic degree $k$. Formally,

$$Lin(k) = \{ L(M) : M \text{ is an nla and } Ndeg(M) = k \}$$

**Theorem 11** $Lin(0) = DLin$, $Lin(k - 1) \subset Lin(k)$ for each $k \in \mathbb{N}^+$ and $\bigcup_{k \in \mathbb{N}} Lin(k) = Lin$.

**Proof.** Let $M$ be an nla. Then $L(M) \in Lin(0)$ iff $Ndeg(M) = 0$ iff $| \delta(q, a) | \leq 1$ iff $M$ is a dla iff $L(M) \in DLin$. Therefore $Lin(0) = DLin$.

Let $M = (Q_L, Q_R, \Sigma, \delta, I, F)$ be an nla such that $L(M) \in Lin(k)$ and $Q = \{ q_1', q_2' \}$ a set of states such that $Q \cap (Q_L \cup Q_R) = \emptyset$. Then $M' = (Q_L \cup Q, Q_R, \Sigma, \delta', I, F)$ where for each $a \in \Sigma$, we have that $\delta'(q,a) = \delta(q,a)$ if $q \in Q_L \cup Q_R$, $\delta'(q_0', a) = Q$ and $\delta'(q_i', a) = \emptyset$ for each $i = 1, \ldots, k$. Clearly, $L(M') = L(M)$ and $Ndeg(M') = Ndeg(M) + 1 = k + 1$. Thus, $L(M) \in Lin(k + 1)$ and therefore $Lin(k) \subseteq Lin(k + 1)$. On the other hand, for each $k \in \mathbb{N}$, the language $L(k) = \{ a^m b^n : m \leq n \leq (k + 1)m \}$ is accepted by the nla of Figure 5 and clearly in $L(k) \in Lin(k)$. Moreover, is evident that it is not possible to construct another nla $M'$, for this same language, with a leaser explicit nondeterministic degree, i.e. such that $Ndeg(M') < k$.

Let $L$ be a linear language. Then by Theorems 4 and 5, the is an nla $M$ such that $L(M) = L$. Let $k = Ndeg(M)$ then $L \in Lin(k)$ and therefore, $L \in \bigcup_{k \in \mathbb{N}} Lin(k)$. So, $\bigcup_{k \in \mathbb{N}} Lin(k) = Lin$. 

---

2 There are other hierarchies varying from $DCFL$ into $CFL$, but based in other machine models, e.g. based on contraction automata [30] and based on restarting automata [16,24,25].
6 Even linear languages

The class of even linear languages was introduced by Amar and Putzolu in [1]. This class contains properly the class of regular language and is properly contained in the class of linear languages. An important characteristic of this class of language is that it allows a solution of the learning problem\(^3\) for some subclasses of even linear languages based on positives examples such as the class of deterministic even linear languages [19].

Basically, a language is even linear if it is generated by an even linear grammar, i.e. a linear grammars where each production of the form \(A \rightarrow uBv\) satisfy \(|u| = |v|\) [19,31]. As is well know, each even linear grammar has a normal form where each production has either the form \(A \rightarrow uBv\) or the form \(A \rightarrow a\), where \(|u| = |v| = 1\) and \(a \in \Sigma \cup \{\lambda\}\) [19].

Let \(M\) be a NLA. \(M\) is an even NLA if their transition diagram is a bipartite graph with \(Q_L\) and \(Q_R\) as their partitions, i.e. if for each \(q \in Q_L\), \(p \in Q_R\) and \(a \in \Sigma\), \(\delta(q,a) \subseteq Q_R\) and \(\delta(p,a) \subseteq Q_L\). For example, the DLA in Figure 4 is an even NLA.

**Proposition 12** Let \(G\) be an even linear grammar. Then there is an even NLA \(M\) such that \(L(G) = L(M)\).

**Proof.** Without loss of generality we can suppose that \(G = \langle V, T, S, P \rangle\) is in the even linear normal form. Let \(G' = \langle V \cup V', T, S, P' \rangle\) be the grammar obtained as follow:

Start with \(V' = P' = \emptyset\). For each production \(A \rightarrow aBb\) in \(P\) add a new variable \(C\) to \(V'\) and the productions \(A \rightarrow aC\) and \(C \rightarrow Bb\) to \(P'\). Finally, add to \(P'\) each production \(A \rightarrow a\) in \(P\). Clearly, \(G'\) is equivalent to \(G\) and is in SLNF.

Now, applying the algorithm in 4, we will has an even NLA equivalent to \(G\).

Conversely,

**Proposition 13** Let \(M\) be an even NLA. Then there is an even linear grammar \(G\) such that \(L(G) = L(M)\).

**Proof.** Applying the algorithm in Theorem 3 to the even NLA \(M\), we will obtain a linear grammar \(G\) with three kind of productions:

\(^3\) The learning problem for a class of formal languages, is the search of "learning procedures" for acquiring grammars on the basis of exposure to evidence about languages in the clase [26].
(i) \( q \to ap \), where \( a \in \Sigma \), \( q \in Q_L \) and \( p \in Q_R \).
(ii) \( p \to qa \), where \( a \in \Sigma \), \( q \in Q_L \) and \( p \in Q_R \).
(iii) \( q \to \lambda \), where \( q \in Q_L \cup Q_R \).

Now, construct a new grammar \( G' = \langle V, T, S, P' \rangle \) from \( G = \langle V, T, S, P \rangle \) as follow:

For each production \( q \to ap \) in \( P \) put in \( P' \) the productions \( q \to ax \) for each \( p \to x \) in \( P \). Analogously, for each production \( p \to qa \) in \( P \) put in \( P' \) the productions \( p \to xa \) for each \( q \to x \) in \( P \).

Clearly, \( G' \) is in the formal norm for even grammar and is equivalent to \( G \) \( \square \)

7 Final remarks

Since, \( \text{nla} \) is a two-read head model which work two way cannot be considered as an automata model in the sense of an abstract family of automata as due by Ginsburg [10]. However, \( \text{nla} \) can be consider as an automata in the more intuitive an general notion, as for example, “An automaton is a device which recognizes or accepts certain elements of \( \Sigma^* \), where \( \Sigma \) is a finite alphabet” [2] or “An automaton is a, construct that possesses all the indispensable features of a digital computer. It accepts inputs, produces output, may have some temporary storage, and can make decisions in transforming the input into the output” [21]

The normal form for linear grammars in section 2 is usefull to turn more easy the proves in section 3 and therefore not intends be an alternative to the well know normal form for the linear grammars.

Recall that the Rosenberg characterization of linear languages is the follow [23,27]: Let \( L \) be a language such that in each word of \( L \), the especial symbol \# occur exactly one times. \( L \) is linear if and only if there is a finite transducer \( M = \langle Q, \Sigma, \Delta, q_0, \delta, F \rangle \) such that their regular translation \( T \) is the set \( \{ (x, y) \in \Sigma^* \times \Delta^* : x\#\tilde{y} \in L \} \), where \( \tilde{y} \) is the reverse of \( y \). In this model the finite transducer receives \( x \) as input and returns \( y \) as output. For the readers could seem that the finite transducer model given by Rosenberg and our \( \text{nla} \) are essentially the same. Nevertheless there are subtle but important differences:

(i) whereas the finite transducer need that the string be split a priory, i.e. before the execution of the automata, in our model the “division of the string” is determined by the \( \text{nla} \) itself. Thus, differently of our model, the finite transducer has no internal mechanism to determine the point of the division of a string in the language. So, these model presuppose an external agent with a meta-knowledge of the language to determine the exact point where each string must be divided.

(ii) finite transducers are nondeterministic and as pointed out by Peter Linz in [21] the role of the nondeterminism in transducers is not clear. Some arguments at this respect also can be found in [6].

(iii) an automata model for a formal language, e.g. \( \text{nla} \), is a formal machine which receives a string as input and then the automata determines if the string is part or not of the language. But, the Rosenberg finite transducer has not this role.
Thus, for each input $w$, this model need first that externally let found the exact point where $w$ must be divided (it could be made by using nondeterminism), then send the first part of the division (e.g. $x$) as input to the transducer model and, finally, verify if the output of the transducer (e.g. $y$) is the desired, i.e. if $xy = w$.

Moreover, $\text{NLA}$ is an intuitive and simple automata model to recognize linear languages which extends in a natural way the NFA in the sense that each NFA can be seen as a NLA. In fact, the simplicity of the proves in this paper are consequence of the simplicity of our model and is perhaps one of the main advantage of NLA on the others models. Another important advantage of the NLA class of automata with respect to others automata models for this class of languages, as early mentioned, is that NLA not consider external agent at models to prepare their inputs.

The contribution of this work was to provide two subclasses of NLA, namely DLA and even NLA, which models the subclasses of deterministic and even linear languages, respectively. In particular, there are different not equivalents proposed for the class of “deterministic” linear languages and here we proved that $\text{DLin}$, i.e. the class of languages accepted by some DLA, contain all those which are origined by a restriction in the linear grammars as considered in [13]. In addition, $\text{DLin}$ is not a proper subset of $\text{DCFL} \cap \text{Lin}$, i.e. the class of linear languages which also are deterministic context free languages. In fact, we conjecture that $\text{DCFL} \cap \text{Lin} \subset \text{DLin}$. The advantage of using the automata models introduced here (DLA and even NLA) is their simplicity with respect other automata models for these classes of languages and in the case of DLA, another advantage is that the class of languages modeled by this class of automata (which are naturally deterministic) is broader than the several classes of deterministic linear languages proposed in the literature. Other minor contributions was provide a characterization of NFA which accept languages also accepted by DLA and a method to obtain this DLA and provide an enumerable hierarchy of linear languages starting by the class DLin and having as limits the class of linear languages.

References


196
Sequent-Based Argumentation

Ofer Arieli 1

School of Computer Science, The Academic College of Tel-Aviv, Israel

Abstract

Logic-based approaches for analyzing and evaluating arguments have been largely studied in recent years, yielding a variety of formal methods for argumentation-based reasoning. In this talk we introduce an abstract, proof theoretical approach to logical argumentation. This is realized in three aspects:

• Arguments are represented by sequents. Sequents can be regarded as specific kinds of judgments and so they are useful for representing logical arguments. This allows us to incorporate sequent-based approaches in argumentation theory, like using sequent calculi for producing arguments in an automated way. Moreover, some restrictions in previous definitions of logical arguments, such as minimality and consistency of support sets are lifted, allowing for a more flexible way of expressing arguments.

• Conflicts between arguments are represented by sequent elimination rules. Interactions between arguments (expressed by attack relations) are represented in terms of Gentzen-style rules of inference. This induces a general and uniform approach not only for introducing arguments, but also for eliminating them.

• Deductions are made by dynamic proof systems. For explicating actual reasoning in an argumentation framework we extend the standard notion of proofs in sequent calculi. Generally, the fact that an argument can be challenged (and possibly withdrawn) by a counter-argument is reflected in dynamic proofs by the ability to consider certain formulas as not derived at a certain stage of the proof even if they were considered derived in earlier stages of the proof. We show how despite of this non-monotonic nature of dynamic derivations one may still draw irreversible conclusions by using them.

Keeping our sequent-based setting generic and modular allows us to accommodate different types of languages and logics, including non-classical ones. We demonstrate the usefulness of our approach by means of various examples, and show that this approach is rich enough to capture a variety of paradigms for handling conflicting arguments.

This is a joint work with Christian Strasser (U. Bochum).

Keywords: structured argumentation, sequent calculi, dynamic derivations.

---

1 Email: oarieli@mta.ac.il

Selected, revised papers will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
Abstract
Nominal rewriting generalises first-order rewriting by providing support for the specification of binding operators. In this paper, we give sufficient conditions for (local) confluence of closed nominal rewriting theories, based on the analysis of rule overlaps. More precisely, we show that closed nominal rewriting rules where all proper critical pairs are joinable are locally confluent. We also show how to refine the notion of rule overlap to derive confluence of the closed rewriting relation. The conditions that we define are easy to check using a nominal matching algorithm.

Keywords: nominal syntax, rewriting, confluence, binding

1 Introduction

Two key properties of rewrite theories are termination (‘the computation is finite’) and confluence (‘it is deterministic’). Termination and confluence are undecidable in general, but decidable criteria do exist that are sufficient, and so can be used to check that a rewrite theory satisfies these properties.

Criteria for guaranteeing confluence of rewriting theories were first investigated in the context of the λ-calculus and abstract rewrite theories in works such as [15], in which the famous Newman’s Lemma was stated: confluence and local confluence

1 Email: anacrismarie@gmail.com. Author supported by a Ph.D. scholarship from CAPES Brazil.
2 Email: ayala@unb.br. Work partially supported by grant CNPq UNIVERSAL 476952/2013-1.
3 Email: maribel.fernandez@kcl.ac.uk. Work partially supported by grant CsF PVE CAPES 146/2012.
4 Web: gabbay.org.uk

Selected, revised papers will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
coincide for terminating rewrite theories. Nowadays this is seen as a combinatorial
property of abstract rewrite theories that strictly depends on Noetherianity, that is,
well-foundedness of the rewrite relation \([11]\).

Without termination, the Critical Pair Lemma, which is the kernel of the famous
Knuth-Bendix completion procedure, guarantees local confluence of term rewriting
theories \([13]\). The most famous sufficient condition for confluence without termina-
tion, giving also rise to a programming discipline, is orthogonality. Orthogonality
essentially avoids ambiguity through two easily verifiable syntactic constraints on the
rewriting rules: left-linearity, that constrains each variable occurring in the left-hand
side of each rule to appear only once, and non-ambiguity, that constrains left-hand
sides of rules to have no overlaps (except for trivial ones, at variable positions or
between a rule and its copy at the root position). With these syntactic restrictions
confluence of orthogonal rewriting theories is guaranteed \([17]\).

Nominal rewriting generalises first-order rewriting by providing support for the
specification of languages with binding operators. In nominal syntax, there are two
kinds of variables: atoms, which are used to represent object-level variables and can
be abstracted but not be substituted, and meta-variables, called simply variables,
which can be substituted but cannot be abstracted. Substitution of a variable
by a term can capture atoms (unlike higher-order theories, where substitution is
non-capturing). The nominal rewriting relation is defined using equivariant nominal
matching, that is, matching modulo \(\alpha\)-equivalence and atom permutations. If rules
are closed, then nominal matching is sufficient to generate the rewrite relation.
Nominal matching is matching modulo \(\alpha\)-equivalence only, and it is efficient (it can
be solved in linear time \([3]\)).

For nominal rewriting theories, the Critical Pair Lemma and confluence of
orthogonal theories were first investigated in \([7]\), where it was shown that the above-
mentioned results extend to the class of uniform nominal rewriting theories, that
is, theories where rules do not generate new atoms. More precisely, in \([7]\) it is
shown that for the class of uniform theories, if all the non-trivial critical pairs are
joinable then the theory is locally confluent, and therefore confluent if it is also
terminating (by Newman’s Lemma). Another sufficient condition for confluence of
uniform theories is orthogonality: if the rules are left-linear and have no non-trivial
critical pairs then the theory is confluent \([7]\). Trivial critical pairs are defined by
overlaps at variable positions, or overlaps at the root between a rule and its copy
(as for first-order rewrite theories). However, overlaps at the root between a rule
and a permuted variant are not trivial. Both of these criteria rely on checking
all non-trivial critical pairs. It is important to check also the overlaps at the root
between a rule and its permuted variants, because if we miss those overlaps the
theory might not be confluent (see the example later in Subsection 1.1).

In \([19]\), the orthogonality condition given in \([7]\) was relaxed, to permit overlaps
at the root between a rule and its permuted copies, but only for uniform rules that
satisfy an additional condition, called \(\alpha\)-stability.

In this paper we give new criteria for (local) confluence of nominal rewriting. We
show that also the conditions in the Critical Pair Lemma can be relaxed if rules
are uniform and \(\alpha\)-stable: if all the non-trivial critical pairs, except possibly those
caused by overlaps at the root between a rule and its permuted variants, are joinable,
then the theory is locally confluent. Moreover, we give a new sufficient condition for
$\alpha$-stability, which is easy to check as it relies simply on nominal matching.

In addition, we give new improved criteria for closed nominal rewriting: it is
sufficient to check the overlaps generated using just one variant of each rule.

Summarising, the main contributions of this paper are:

(i) We relax the conditions in the Critical Pair Lemma for uniform rewriting rules
that are $\alpha$-stable: it is not necessary to consider critical pairs generated by
overlaps at the root between a rule and a permuted variant. See Subsection 4.1.

(ii) We show that closedness is a sufficient condition for $\alpha$-stability. Since closedness
is easy to check (by simply solving a nominal matching problem), we get an
easy to check condition for $\alpha$-stability. See Subsection 4.2.

(iii) We show that for closed rewriting, the criteria can be relaxed even more: it
is sufficient to check overlaps between freshened versions of rules; overlaps
between permuted variants of rules (at the root or otherwise) do not need to
be considered at all. See Section 5.

1.1 Related work

First-order rewriting systems and the $\lambda$-calculus provide two useful notions of terms
and reduction. However, both have limitations, which motivated extensions such as
higher-order rewriting systems (see, e.g., [12,14]). Nominal rewriting systems are
at an intermediate level between higher-order rewriting systems and their explicit
substitution versions, which implement in a first-order setting the capture-avoiding
substitution operation together with $\alpha$-conversion. For the latter, indices and rewrite
rules are used to deal with the management of bound variables (see, e.g., [18]). Using
nominal rewriting, we can specify capture-avoiding substitutions without the need
to manage indices, since names and $\alpha$-equivalence are primitive notions.

Two notions of orthogonality have been defined for nominal rewriting in previous
work. In [7], orthogonality has been defined as left-linearity and absence of non-trivial
critical pairs, and it was shown that this is a sufficient condition for confluence
of uniform rewrite rules. The definition of orthogonality was relaxed in [19] to
allow overlaps at the root between permuted variants of rules; however, this weaker
definition does not ensure confluence of uniform rules: a rule such as $\vdash f(X) \rightarrow f([a]X)$
is orthogonal according to [19] but it is not confluent; for example, $f(X) \rightarrow f([a]X)$
and $f(X) \rightarrow f([b]X)$, where $\not\vdash f([a]X) \approx_\alpha f([b]X)$. Note that here there are non-
joinable critical pairs generated by overlaps at the root between permuted variants of
the rule. With the additional requirement of $\alpha$-stability, confluence is guaranteed [19].

A sufficient condition for $\alpha$-stability was given in [19], called “abstract skeleton
preserving” (ASP). This is a strong restriction: it only allows identity permutations
to be suspended on variables, and it requires the use of different atoms in nested
abstractions. Here we show that closedness, which does not impose such restrictions
and can be checked simply by solving a nominal matching problem, is a sufficient
condition for $\alpha$-stability. In addition, for closed rewriting the criteria for confluence
can be simplified, by checking only overlaps of freshened rules. Closedness and the
ASP criterion are complementary in the sense that ASP does not imply closedness
and closedness does not imply ASP.
2 Syntax

Fix disjoint countably infinite collections of atoms, unknowns (or variables), and term-formers (or function symbols). We write $\mathcal{A}$ for the set of atoms; $a, b, c, \ldots$ will range over distinct atoms. $X, Y, Z, \ldots$ will range over distinct unknowns. $f, g, \ldots$ will range over distinct term-formers. We assume that to each $f$ is associated an arity $\alpha \geq 0$. A signature $\Sigma$ is a set of term-formers with their arities.

**Definition 2.1** A permutation $\pi$ is a bijection on atoms such that $\text{nontriv}(\pi) = \{a \mid \pi(a) \neq a\}$ is finite. We write $(a \ b)$ for the switching permutation that maps $a$ to $b$, $b$ to $a$ and all other $c$ to themselves, and $\text{id}$ for the identity permutation, so $\text{id}(a) = a$. The notation $\pi \circ \pi'$ is used for functional composition of permutations, so $(\pi \circ \pi')(a) = \pi(\pi'(a))$, and $\pi^{-1}$ for inverse, so $\pi(a) = b$ if and only if $a = \pi^{-1}(b)$.

Permutations are represented by lists of swappings; thus, composition is list concatenation, and the inverse is obtained simply by reversing the list.

**Definition 2.2** Define (nominal) terms inductively by:

$$s, t, l, r, u ::= a \mid \pi \cdot X \mid [a] \equiv f(t_1, \ldots, t_n)$$

Call $\pi \cdot X$ a (suspended) variable and $[a] \equiv$ an (atom-)abstraction; it represents $\lambda x.e$ or $\forall x.\varphi$ in expressions like $\lambda x.e$ or $\forall x.\varphi$. We write $\equiv$ for syntactic identity.

**Definition 2.3** Define $\pi \cdot t$ a permutation action by:

$$\pi \cdot a \equiv \pi(a) \quad (\pi \cdot \pi') \cdot X \equiv (\pi \circ \pi') \cdot X \quad \pi \cdot [a] \equiv [\pi(a)](\pi \cdot t) \quad \pi \cdot f(t_1, \ldots, t_n) \equiv f(\pi \cdot t_1, \ldots, \pi \cdot t_n)$$

A substitution (on unknowns), ranged over by $\theta, \sigma, \ldots$, is a partial function from unknowns to terms with finite domain. We write $\text{id}$ for the substitution with $\text{dom}(\text{id}) = \emptyset$ (it will always be clear whether we mean $\text{id}$ the identity substitution’ or ‘$\text{id}$ the identity permutation’). If $X \not\in \text{dom}(\sigma)$ then $\sigma(X)$ denotes $\text{id} \cdot X$.

Define $t\sigma$ an (unknowns) substitution action by:

$$a\sigma \equiv a \quad (\pi \cdot X)\sigma \equiv \pi \cdot X \quad (X \not\in \text{dom}(\sigma))$$

$$([a])\sigma \equiv [a](t\sigma) \quad (\pi \cdot X)\sigma \equiv \pi \cdot \sigma(X) \quad (X \in \text{dom}(\sigma))$$

$$f(t_1, \ldots, t_n)\sigma \equiv f(t_1\sigma, \ldots, t_n\sigma)$$

If $\sigma$ and $\theta$ are substitutions, $\sigma \circ \theta$ maps each $X$ to $(X\theta)\sigma$.

**Definition 2.4** The set $\text{Pos}(t)$ of positions of a term $t$ is defined below. Note that $\epsilon$ is the only position in atoms and variables.

$$\epsilon \in \text{Pos}(t) \quad p \in \text{Pos}(t) \quad (p)\sigma \quad p \in \text{Pos}(t_i) \quad (\sigma) \quad p \in \text{Pos}(t_i) \quad (\sigma) \quad \frac{p \in \text{Pos}(t_i)}{i \cdot p \in \text{Pos}(f(t_1, \ldots, t_i, \ldots, t_n))} \quad (p)$$

Call $t|_p$ a subterm of $t$ at position $p$ when

$$t|_p = t \quad [a] t|_p = t \quad f(t_1, \ldots, t_i, \ldots, t_n)|_p = t|_p \quad (1 \leq i \leq n)$$

If $p \in \text{Pos}(s)$, then $s[p \leftarrow t]$ denotes the replacement of $s|_p$ by $t$ in $s$. 

202
\[
\begin{align*}
\Delta \vdash a \# b & \quad (\# \text{ab}) \\
\Delta \vdash a \# \pi \cdot X & \quad (\# X) \\
\Delta \vdash a \# t_1 \cdots \Delta \vdash a \# t_n & \quad (\# f) \\
\Delta \vdash b \# t & \quad \Delta \vdash (b \cdot t) \equiv_a u \quad (\equiv_a \text{b}) \\
\Delta \vdash t \equiv_a u & \quad \Delta \vdash [a] t \equiv_a [a] u \quad (\equiv_a \text{a}) \\
\Delta \vdash a \# X & \quad \Delta \vdash \pi \cdot X \equiv_a \pi' \cdot X \quad (\equiv_a X) \\
\Delta \vdash f(t_1, \ldots, t_n) \equiv_a f(u_1, \ldots, u_n) & \quad (\equiv_a f)
\end{align*}
\]

**Figure 1:** Freshness and α-equality

**Definition 2.5** A freshness (constraint) is a pair \( a \# t \) of an atom \( a \) and a term \( t \). We call a freshness of the form \( a \# X \) primitive, and a finite set of primitive freshes a freshness context. \( \Delta, \Gamma \) and \( \nabla \) will range over freshness contexts.

We denote by \( \nabla \) the set \( \{ a \# \sigma(X) \mid a \# X \in \nabla \} \) of freshness constraints.

A freshness judgement is a tuple \( \Delta \vdash a \# t \) of a freshness context and a freshness constraint. An \( \alpha \)-equivalence judgement is a tuple \( \Delta \vdash s \equiv_a t \) of a freshness context and two terms. The derivable freshness and \( \alpha \)-equivalence judgements are defined by the rules in Figure 1, where \( ds(\pi, \pi') = \{ a \in A \mid \pi(a) \neq \pi'(a) \} \). We call \( ds(\pi, \pi') \) the difference set of permutations \( \pi \) and \( \pi' \).

**Definition 2.6** The functions \( \text{atms}(t) \) and \( \text{unkn}(t) \) will be used to compute the set of atoms and unknowns in a term, respectively. They are defined by:

\[
\begin{align*}
\text{atms}(a) &= \{ a \} \\
\text{atms}([a] t) &= \text{atms}(t) \cup \{ a \} \\
\text{atms}(\pi \cdot X) &= \text{nontriv}(\pi) \\
\text{atms}(f(t_1, \ldots, t_n)) &= \bigcup_i \text{atms}(t_i) \\
\text{unkn}(a) &= \emptyset \\
\text{unkn}([a] t) &= \text{unkn}(t) \\
\text{unkn}(\pi \cdot X) &= \{ X \} \\
\text{unkn}(f(t_1, \ldots, t_n)) &= \bigcup_i \text{unkn}(t_i)
\end{align*}
\]

**3 Nominal Rewriting**

**Definition 3.1** A rewrite judgement is a tuple \( \nabla \vdash l \rightarrow r \) of a freshness context and two terms. We may write ‘\( \emptyset \vdash l \rightarrow r \)’ as ‘\( l' \rightarrow r \)’.

A rewrite theory \( R = (\Sigma, Rw) \) is a pair of a signature \( \Sigma \) and a possibly infinite set of rewrite judgements \( Rw \) in that signature; we call these rewrite rules.

A rewrite rule \( \nabla \vdash l \rightarrow r \) is left-linear if each unknown occurs at most once in \( l \).

**Definition 3.2** Define \( t^\pi \) the meta-action of \( \pi \) on \( t \) by:

\[
\begin{align*}
\pi(a) &= a^\pi \quad (\rho \cdot X)^\pi = \rho^\pi \cdot X \\
([a] t)^\pi &= [a^\pi] t^\pi \\
f(t_1, \ldots, t_n)^\pi &= f(t_1^\pi, \ldots, t_n^\pi),
\end{align*}
\]

where \( id^\pi = id \) and \( ((a \cdot b) \circ \rho)^\pi = (\pi(a) \pi(b)) \circ \rho^\pi \).
Extend the meta-action to contexts by $\nabla^π = \{ π(a)\#X \mid a\#X \in \nabla \}$.

The meta-action of permutations affects only atoms in terms (it does not suspend on variables, in contrast with the permutation action of Definition 2.3). We use it to define the equivariant closure of a set of rules, needed to generate the rewrite relation (Definition 3.4; see [7] for more details).

**Definition 3.3** The equivariant closure of a set $Rw$ of rewrite rules is the closure of $Rw$ by the meta-action of permutations, that is, it is the set of all the permutative variants of rules in $Rw$. We write $eq\cdot closure(Rw)$ for the equivariant closure of $Rw$.

Below we write $\Delta \vdash (\phi_1, \ldots, \phi_n)$ for the judgements $\Delta \vdash \phi_1$, $\ldots$, $\Delta \vdash \phi_n$.

**Definition 3.4** Nominal rewriting: Let $R = (\Sigma, Rw)$ be a rewrite theory. The one-step rewrite relation $\Delta \vdash s \stackrel{R}{\rightarrow} t$ is the least relation such that for every $(\nabla \vdash l \rightarrow r) \in Rw$, position $p$, permutation $\pi$, and substitution $\theta$,

$$\Delta \vdash (\nabla^\pi \theta, \ s\vert_p \approx_\alpha l^\pi \theta, \ s\vert_p \leftarrow r^\pi \theta \approx_\alpha t) \quad (Rew_{\nabla\vdash l\rightarrow r})$$

The notation $\Delta \vdash s \rightarrow_{(R, p, \pi, \theta)} t$ highlights the fact that the rewrite step from $s$ to $t$ occurs with some specific rule $R$, position $p$, permutation $\pi$ and substitution $\theta$, under the freshness context $\Delta$.

The rewrite relation $\Delta \vdash_R s \rightarrow t$ is the reflexive transitive closure of the one-step rewrite relation, that is, the least relation that includes the one-step rewrite relation and such that:

- for all $\Delta$ and $s$: $\Delta \vdash s \approx_\alpha s'$ implies $\Delta \vdash_R s \rightarrow s'$; and
- for all $\Delta$, $s$, $t$, $u$: $\Delta \vdash_R s \rightarrow t$ and $\Delta \vdash_R t \rightarrow u$ implies $\Delta \vdash_R s \rightarrow u$.

If $\Delta \vdash_R s \rightarrow t$ holds, we say that $s$ rewrites to $t$ in the context $\Delta$.

The rewrite relation is defined in a freshness context since it takes into account $\alpha$-equivalence, which depends on freshness information for the term unknowns.

**Example 3.5** The following rewrite theory, using a signature containing termformers $\lambda$ of arity 1, and $app$ and $subst$ of arity 2, defines $\beta$-reduction for the $\lambda$-calculus. Below, application is denoted by juxtaposition and $subst([a]X, Y)$ is written $X[a \mapsto Y]$ as usual (syntactic sugar). In this theory, we can derive $\vdash_R (\lambda[a]a)Y \rightarrow Y$ and also $a\#Z \vdash_R (\lambda[a]Z)Y \rightarrow Z$.

(Beta) $\quad \vdash (\lambda[a]X)Y \rightarrow X[a \mapsto Y]

\sigma_{app} \quad \vdash (XX')[a \mapsto Y] \rightarrow X[a \mapsto Y]X'[a \mapsto Y]

\sigma_{var} \quad \vdash a[a \mapsto X] \rightarrow X

\sigma_{lam} \quad b\#Y \vdash (\lambda[b]X)[a \mapsto Y] \rightarrow \lambda[b](X[a \mapsto Y])

\sigma_{e} \quad a\#X \vdash X[a \mapsto Y] \rightarrow X$

**Definition 3.6** A rewrite theory $R$ is terminating if there are no infinite rewriting sequences. It is locally confluent if $\Delta \vdash s \stackrel{R}{\rightarrow} u$ and $\Delta \vdash s \stackrel{R}{\rightarrow} v$ implies that there exists $w$ such that $\Delta \vdash_R u \rightarrow w$ and $\Delta \vdash_R v \rightarrow w$. It is confluent when, if $\Delta \vdash_R s \rightarrow t$ and $\Delta \vdash_R s \rightarrow t'$, then $u$ exists such that $\Delta \vdash_R t \rightarrow u$ and $\Delta \vdash_R t' \rightarrow u$. 

204
We call the situation \( \Delta \vdash s \xrightarrow{R} u \) and \( \Delta \vdash s \xrightarrow{R} v \) a peak.

**Remark 3.7** Since the definition of the rewriting relation generated by a rewrite theory \( R \) takes into account permuted variants of rules (via the use of the permutation \( \pi \) in the one-step rewrite relation), it is not necessary to include permuted variants of rules in \( Rw \). For convenience, in the rest of the paper we assume that for any \( R \in Rw \), if \( R \) and \( R^\pi \) are both in \( Rw \) then \( \pi = id \); in other words, \( Rw \) does not contain permuted variants of the same rule.

According to Definition 3.4, to generate a rewrite step we need to solve an equivariant matching problem, that is, we need to find a permutation and a substitution such that \( \Delta \vdash s \mid_p \approx_{\alpha} l \theta \). This problem is decidable, but NP-complete [4]. However, for **closed rules** [7], a simpler matching problem of the form \( \Delta \vdash s \mid_p \approx_{\alpha} l \theta \), called nominal matching [20], suffices to generate the rewrite relation. Nominal matching is decidable and unitary [20] and efficient (it can be solved in linear time [3,2]).

Closed rules roughly correspond to rules without free atoms, where rewriting cannot change the binding status of an atom. They are the counterpart of rules in standard higher-order rewriting formats (see [6]). Below we first recall the definition of nominal matching and then give a structural definition and an operational characterisation of closed terms.

**Definition 3.8** A **term-in-context** is a pair \( \Delta \vdash t \) of a freshness context and a term. A **nominal matching problem** is a pair of terms-in-context \((\nabla \vdash l) \approx (\Delta \vdash s)\) where \( \text{unkn}(\nabla \vdash l) \cap \text{unkn}(\Delta \vdash s) = \emptyset \).

A **solution** to this problem is a substitution \( \sigma \) such that \( \Delta \vdash \nabla \sigma, \Delta \vdash l \sigma \approx_{\alpha} s \), and \( \text{dom}(\sigma) \subseteq \text{unkn}(\nabla \vdash l) \).

The following structural definition of closedness is taken from [6]; see also [5].

**Definition 3.9** Call a term-in-context \( \Delta \vdash t \) **closed** when

(i) every occurrence of an atom \( a \in \text{atms}(t) \) is in the scope of an abstraction of \( a \);

(ii) if \( \pi \cdot X \) occurs in the scope of an abstraction of \( \pi \cdot a \) then any occurrence of \( \pi' \cdot X \) occurs in the scope of an abstraction of \( \pi' \cdot a \) or \( a \# X \in \Delta \);

(iii) for any pair \( \pi_1 \cdot X, \pi_2 \cdot X \) occurring in \( t \), and \( a \in \text{ds}(\pi_1, \pi_2) \), if \( a \) does not occur in the scope of an abstraction in one of the occurrences then \( a \# X \in \Delta \).

Call \( R = (\nabla \vdash l \rightarrow r) \) **closed** when \( \nabla \vdash (l, r) \) is closed\(^5\).

It is easy to check whether a term is closed, using nominal matching and a freshened variant of the term as shown in [7] (see Proposition 3.11 below).

**Definition 3.10** A **freshened variant** \( t^o \) of a nominal term \( t \) is a term with the same structure as \( t \), except that the atoms and unknowns are replaced by ‘fresh’ atoms and unknowns (so they are not in \( \text{atms}(t) \) and \( \text{unkn}(t) \), and perhaps are also fresh with respect to some atoms and unknowns from other syntax, which we will always specify). We omit an inductive definition.

\(^5\) Here we use pair as a term former and apply the definition above.
Similarly, if $\nabla$ is a freshness context then $\nabla^*$ denotes a freshened variant of $\nabla$ (so if $a \# X \in \nabla$ then $a^* \# X^* \in \nabla^*$, where $a^*$ and $X^*$ are chosen fresh for the atoms and unknowns appearing in $\nabla$).

We may extend this to other syntax, like equality and rewrite judgements.

Note that if $\nabla^* \triangleright l \rightarrow r^*$ is a freshened variant of $\nabla \triangleright l \rightarrow r$ then $\text{unkn}(\nabla^* \triangleright l \rightarrow r^*) \cap \text{unkn}(\nabla \triangleright l \rightarrow r) = \emptyset$.

**Proposition 3.11** A term-in-context $\nabla \triangleright l$ is closed if and only if there exists a solution for the matching problem

$$\text{atsms}(\nabla^*,l') \approx (\nabla, \text{atsms}(\nabla^*,l') \# \text{unkn}(\nabla,l) \triangleright l).$$

(1)

**Definition 3.12** Given a rewrite rule $R = (\nabla \triangleright l \rightarrow r)$ and a term-in-context $\Delta \triangleright s$, write $\Delta \triangleright s \xrightarrow{R_\nabla} t$ when there is some $R'$ a freshened variant of $R$ (so, fresh for $R$, $\Delta$, $s$, and $t$), position $p$ and substitution $\theta$ such that

$$\Delta, \text{atsms}(R') \# \text{unkn}(\Delta,s,t) \triangleright (\nabla^* \theta, s \approx_{\alpha} l' \theta, s[p \leftarrow r^* \theta] \approx_{\alpha} t).$$

(2)

We call this (one-step) closed rewriting.

The **closed rewrite relation** $\Delta \triangleright R \Delta \triangleright s \rightarrow_c t$ is the reflexive transitive closure of the one-step closed rewrite relation (as in Definition 3.4, but notice the extended freshness context).

**Example 3.13** Any rule with free atoms, such as $\triangleright f(a,a) \rightarrow a$, is not closed (it is impossible to match it with a freshened variant). The rule $R = [a]f(a,X) \rightarrow 0$ is closed, since taking a freshened version $R_1 = [b]f(b,Y) \rightarrow 0$, it is possible to solve the matching problem ($\triangleright ([b]f(b,Y),0)) \approx (b\#X \triangleright ([a]f(a,X),0))$ with the substitution $\sigma = [Y \leftrightarrow (a\ b) \cdot X]$. Notice that $b\#X \triangleright [b]f(,(a\ b) \cdot X)$  $\approx [a]f(a,X)$.

We refer to [7] for more examples and properties of closed rewriting.

To compute overlaps of rules, we use a nominal unification algorithm [20].

**Definition 3.14** A **nominal unification problem** is a set of freshness constraints and pairs of terms of the form $s \approx_{\gamma} u$. A unification problem

$$\{a_1 \# t_1, \ldots, a_k \# t_k, s_1 \approx_{\gamma} u_1, \ldots, s_m \approx_{\gamma} u_m\}$$

is unifiable if there exists a **solution** $(\Gamma, \theta)$ (freshness context and substitution) such that $\Gamma \triangleright (a_1 \# t_1 \theta, \ldots, a_k \# t_k \theta, s_1 \theta \approx_{\alpha} u_1 \theta, \ldots, s_m \theta \approx_{\alpha} u_m \theta)$. In this case, $(\Gamma, \theta)$ is said to be a **unifier** for the problem.

Nominal unification is decidable and unitary, that is, if there is a solution for a nominal unification problem there exists a most general one.

## 4 Confluence of Nominal Rewriting

In this section we consider two well-known criteria for confluence of first-order rewriting based on the notion of overlapping rewrite steps [1]. They can be extended to nominal rewrite theories, but it is necessary to add some conditions.
4.1 Critical Pair Criterion and Orthogonality

The notion of overlap has been extended from the first-order setting to the nominal rewriting setting [7]. In the first-order case, overlaps are computed by unification of a left-hand side of a rule $R_1$ with a non-variable subterm of the left-hand side of a rule $R_2$ (which could be a copy of $R_1$ with renamed variables, in which case the subterm must be strict, that is, overlaps at the root between a left-hand side and its copy are not considered). In the case of nominal rules, since the nominal rewrite relation is generated by the equivariant closure of a set of rules, we must take into account permuted variants of rules, and use nominal unification instead of first-order unification. Definition 4.1 follows [7] and takes into account this situation.

**Definition 4.1 (Overlaps and CPs)** Let $R_i = \overrightarrow{\gamma_i \iota_i \rightarrow r_i}$ ($i = 1, 2$) be copies of rewrite rules in $\text{eq-closure}(R\omega)$ (so $R_1$ and $R_2$ could be copies of the same rule), where $\text{unkn}(R_1) \cap \text{unkn}(R_2) = \emptyset$, as usual. If the nominal unification problem $\overrightarrow{\gamma_1 \cup \gamma_2 \cup \{\iota_2 \gamma \approx? \iota_1 \nu\}}$ has a most general solution $\langle \Gamma, \theta \rangle$ for some position $p$, then we say that $R_1$ overlaps with $R_2$, and we call the pair of terms-in-context $\Gamma \vdash \langle r_1 \theta, \iota_1 \theta[p \leftarrow r_2 \theta]\rangle$ a critical pair. If $p$ is a variable position, or if $R_1$ and $R_2$ are identical modulo renaming of variables and $p = \epsilon$, then we call the overlap and critical pair trivial, otherwise we call it non-trivial.

The critical pair $\Gamma \vdash \langle r_1 \theta, \iota_1 \theta[p \leftarrow r_2 \theta]\rangle$ is joinable if there is a term $u$ such that $\Gamma \vdash r_1 \theta \rightarrow u$ and $\Gamma \vdash \iota_1 \theta[p \leftarrow r_2 \theta] \rightarrow u$.

We distinguish between different kinds of overlaps and critical pairs:

**Definition 4.2 (Permutative Overlaps and CPs)** Let $R_i = \overrightarrow{\gamma_i \iota_i \rightarrow r_i}$ ($i = 1, 2$) be copies of rewrite rules in $\text{eq-closure}(R\omega)$, such that there is an overlap. If $R_2$ is a copy of $R_1^\pi$, we say that the overlap is permutative. We call a permutative overlap at the root position root-permutative. We call an overlap that is not trivial and not root-permutative proper. We use the same terminology to classify critical pairs; e.g. we call a critical pair generated by a permutative overlap permutative.

A permutative overlap indicates that there is a critical pair generated by a rule and one of its permuted variants.

Note that only the root-permutative overlaps where $\pi$ is $\text{id}$ are trivial. While overlaps at the root between variable-renamed versions of first-order rules can be discarded (they generate equal terms), in nominal rewriting we must also consider overlaps at the root between permuted variants of rules. Indeed, they do not necessarily produce the same result, as the following example shows (see also [19]).

**Example 4.3** Consider $R = (\vdash f(X) \rightarrow f([a]X))$. There is an overlap at the root between this rule and its variant $R^{(a \ b)} = (\vdash f(X) \rightarrow f([b]X))$, i.e., a root-permutative overlap, which is not trivial. It generates the critical pair $\vdash (f([a]X), f([b]X))$. Note that the terms $f([a]X)$ and $f([b]X)$ are not $\alpha$-equivalent. This theory is not confluent; we have for instance:

\[
\begin{array}{c}
\text{f(a)} \\
\begin{array}{c}
\begin{array}{c}
\text{f([a]X)} \\
\text{f([b]X)}
\end{array}
\end{array}
\end{array}
\]

207
Definition 4.4 introduces uniformity. In [7] a Critical Pair Lemma was proved for uniform nominal rewrite theories, that absence of non-trivial critical pairs implies local confluence; confluence follows by Newman’s Lemma if the theory is terminating. Uniformity features in this paper in Theorems 4.6, 5.3, and 5.4. Intuitively, uniformity means that if \( a \) is not free in \( s \) and \( s \) rewrites to \( t \) then \( a \) is not free in \( t \).

**Definition 4.4 (Uniformity)** We call a nominal rewrite theory \( R = (\Sigma, Rw) \) uniform [7] when if \( \Delta \vdash_R s \rightarrow t \) and \( \Delta, \Delta' \vdash a\#s \) for some \( \Delta' \), then \( \Delta, \Delta' \vdash a\#t \).

We stress that in the Critical Pair Lemma of [7], the assumption is that only trivial critical pairs may be present. Absence of proper critical pairs is not a sufficient condition for local confluence, even if the theory is uniform: the rule in Example 4.3 is uniform. However, with an additional condition, we can prove that uniform rewrite theories without proper critical pairs are locally confluent. We recall below the notion of \( \alpha \)-stability defined in [19].

**Definition 4.5 (\( \alpha \)-stability)** Call a rewrite rule \( \Delta \vdash l \rightarrow r \) \( \alpha \)-stable when, for all \( \Delta, \pi, \sigma, \sigma' \), \( \Delta \vdash l \sigma \approx_{\alpha} l \pi \sigma' \) implies \( \Delta \vdash r \sigma \approx_{\alpha} r \pi \sigma' \).

A rewrite theory \( R = (\Sigma, Rw) \) is \( \alpha \)-stable if every rule in \( Rw \) is \( \alpha \)-stable.

Note that \( \alpha \)-stability is hard to check in general, because of the quantification over all \( \sigma \). The notion of \( \alpha \)-stability is related to the notion of closedness (Definition 3.9): we show in Section 4.2 that closed rules are \( \alpha \)-stable. The reverse implication does not hold: for example, the rule \( \vdash f(a) \rightarrow a \) is \( \alpha \)-stable but not closed.

**Theorem 4.6 (Critical Pair Lemma for uniform \( \alpha \)-stable theories)** Let \( R = (\Sigma, Rw) \) be a uniform rewrite theory where all the rewrite rules in \( Rw \) are \( \alpha \)-stable. If every proper critical pair is joinable, then \( R \) is locally confluent.

**Proof** By case analysis. There are four kinds of peaks:

- If the rewrite steps occur at disjoint positions, then the peak is trivially joinable by applying the same rules, permutations and substitutions.
- If the peak is an instance of a proper critical pair (joinable by assumption) then it is joinable since the rewriting relation is compatible with instantiation, as shown in [7], Theorem 49 (for closedness implies uniformity).
- If the peak is generated by an overlap at a variable position, without loss of generality we assume \( \Delta \vdash s \approx_{\alpha} l_1^{\pi_1} \sigma_1 \) and \( s \) occurs inside \( l_2^{\pi_2} \sigma_2 \) under an instance of an unknown \( (\pi^{\pi_2} \cdot X)\sigma_2 \) (see Figure 2). Then we can change the action of \( \sigma_2 \) over \( X \), replacing \( s \) by \( t \), such that \( \Delta \vdash t \approx_{\alpha} r_1^{\pi_1} \sigma_1 \), as it is done in the first-order case. Here we rely on the assumption of uniformity, which ensures that no free atoms are introduced by the rewrite step, therefore no freshness constraint will be violated when replacing \( s \) by \( t \). Again, by Theorem 49 in [7], if a peak diverges through this overlap, then local confluence holds for the rewriting relation is compatible with instantiation.
- If there is a root-permutative overlap, then joinability follows by the assumption of \( \alpha \)-stability.

We now consider a confluence criterion based on orthogonality.
Definition 4.7 Call a rewrite theory $R = (\Sigma, Rw)$ orthogonal when all the rules in $Rw$ are left-linear and there are no non-trivial critical pairs.

Call $R = (\Sigma, Rw)$ quasi-orthogonal when all rules are left-linear and there are no proper overlaps.

So orthogonal theories are left-linear and can have trivial overlaps only, whereas quasi-orthogonal theories are left-linear and can have trivial overlaps and root-permutative overlaps (Definition 4.2).

Orthogonal theories were defined in [7]. Quasi-orthogonal theories were defined in [19] and called orthogonal (we changed the name here to avoid confusion).

For uniform nominal rewrite theories, orthogonality implies confluence [7]. Quasi-orthogonality is not sufficient for confluence of uniform theories; see Example 4.3. If a theory is uniform, quasi-orthogonal, and $\alpha$-stable, then it is confluent [19]. The proof structure is the same as in [1,16,7], i.e., the diamond property is proved for a parallel rewriting relation $\Delta \vdash \_ \Rightarrow \_ \ (\text{simultaneous rewrite steps at disjoint positions})$. As the reflexive transitive closure of the parallel relation $\Delta \vdash \_ \Rightarrow \_ \$ is equal to $\Delta \vdash \_ \Rightarrow \_ \text{ (simultaneous rewrite steps)}$, then diamond-property of $\Delta \vdash \_ \Rightarrow \_ \$ is sufficient to prove confluence of $\Delta \vdash \_ \Rightarrow \_ \$. Figure 3 illustrates the proof. The overlaps may only occur under instances of variables or be root-permutative. So, it is necessary to join the subterms which are at positions marked with "*" in the terms $t_1$ and $t_2$, in order to obtain the common term $u$. Observe the subterm that occurs in the middle of term $s$, in Figure 3. This subterm changes in term $t_1$ and $t_2$ at the same position. In a root-permutative overlap, by $\alpha$-stability, the corresponding subterms in $t_1$ and $t_2$ are $\alpha$-equivalent. If the overlap is at a variable position, then the proof proceeds as in the other case, using the assumption of uniformity to ensure that when we
change the substitution, the rewrite step is still possible. In Figure 3, the instances of variables are represented by pentagons.

4.2 Criterion for $\alpha$-stability

This section presents closedness as a sufficient condition for $\alpha$-stability. Closedness is easy to check using a nominal matching algorithm (see Proposition 3.11).

First, we need an auxiliary technical lemma, which states that if two substitutions coincide (modulo $\alpha$) for all the unknowns in a term, the corresponding instances are $\alpha$-equivalent. We omit the inductive proof.

Lemma 4.8 $\Delta \vdash t \sigma \approx_\alpha t \theta \iff \forall X \in \text{unkn}(t), \Delta \vdash X \sigma \approx_\alpha X \theta$.

Lemma 4.9 If $R$ is a closed rule, then $R$ is $\alpha$-stable.

Proof It is sufficient to prove the following property:

$$R = \nabla \vdash l \to r \text{ closed}$$

$$\Delta \vdash s \approx_\alpha l \sigma \to r \sigma$$

$$\Delta \vdash s \approx_\alpha l^\pi \sigma' \to r^\pi \sigma'$$

then $\Delta \vdash r \sigma \approx_\alpha r^\pi \sigma'$.

Since $R$ is closed, we have

$$\begin{cases} (\nabla \vdash (l^\pi, r^\pi)) \approx_\alpha (\nabla, \text{atms}(R^\pi) \# \text{unkn}(R) \vdash (l, r)) \\ (\nabla \vdash (l^\pi, r^\pi)) \approx_\alpha (\nabla^\pi, \text{atms}(R^\pi) \# \text{unkn}(R) \vdash (l^\pi, r^\pi)) \end{cases}$$

solvable with solutions $\theta$ and $\theta^\pi$, respectively. Hence, we have:

- $\nabla, \text{atms}(R^\pi) \# \text{unkn}(R) \vdash \nabla^\pi \theta$
- $\nabla, \text{atms}(R^\pi) \# \text{unkn}(R) \vdash (l^\pi \theta, r^\pi \theta) \approx_\alpha (l, r)$
- $\nabla^\pi, \text{atms}(R^\pi) \# \text{unkn}(R) \vdash \nabla^\pi \theta^\pi$
- $\nabla^\pi, \text{atms}(R^\pi) \# \text{unkn}(R) \vdash (l^\pi \theta^\pi, r^\pi \theta^\pi) \approx_\alpha (l^\pi, r^\pi)$
- $\Delta \vdash \nabla \sigma, \nabla^\pi \sigma'$
- $\Delta \vdash l \sigma \approx_\alpha l^\pi \sigma' \implies \Delta, \text{atms}(R^\pi) \# \text{unkn}(R \sigma) \vdash l^\pi \theta \sigma \approx_\alpha l^\pi \theta^\pi \sigma'$
From the part \((\Rightarrow)\) of Lemma 4.8, it follows that \(\forall X \in \text{unkn}(\Gamma^t) : \Delta, \text{atms}(\Gamma^t) \# \text{unkn}(\Gamma) \vdash X\theta\sigma \approx_{\alpha} X\theta_\sigma'\).

Since \(\text{unkn}(\Gamma^t) \subseteq \text{unkn}(\Gamma)\), we can use the part \((=)\) of Lemma 4.8 and obtain what we want to demonstrate:
\[
\begin{align*}
\Delta, \text{atms}(\Gamma^t) \# \text{unkn}(\Gamma) & \vdash r^\theta\sigma \approx_{\alpha} r\sigma \\
\Rightarrow & \\
\Delta, \text{atms}(\Gamma^t) \# \text{unkn}(\Gamma) & \vdash r^\theta\sigma' \approx_{\alpha} r^\pi\sigma'
\end{align*}
\]

Since the atoms in \(\text{atms}(\Gamma^t)\) do not occur in \(r\sigma, r^\pi\sigma'\), we can strengthen the previous judgement, taking only \(\Delta\) as context. \(\square\)

5 Better Criteria for Confluence of Closed Rewriting

In this section we study confluence of closed rewriting (Definition 3.12). Closed rewriting uses freshened versions of rules and nominal matching, instead of the computationally more expensive equivariant matching used in Definition 3.4. Closed rewriting is complete for equational reasoning if the axioms are closed [8].

The following definitions of fresh overlap and fresh critical pair will be used to derive sufficient conditions for confluence of closed rewriting.

Definition 5.1 (Fresh Overlaps and CPs) Let \(R_i = \nabla_i \vdash l_i \rightarrow r_i\ (i = 1, 2)\) be freshened versions of rewrite rules in \(Rw\) (\(R_1\) and \(R_2\) could be two freshened versions of the same rule), where \(\text{unkn}(R_1) \cap \text{unkn}(R_2) = \emptyset\), as usual. If the nominal unification problem \(\nabla_1 \cup \nabla_2 \cup \{l_2 \gamma \approx_{\tau} l_1|_{p}\}\) has a most general solution \(\langle \Gamma, \theta \rangle\) for some position \(p\), then we say that \(R_1\) \textbf{fresh overlaps} with \(R_2\), and we call the pair of terms-in-context \(\Gamma \vdash (r_1\theta, l_1\theta[p \leftarrow r_2\theta])\) a \textbf{fresh critical pair}.

If \(p\) is a variable position, or if \(R_1\) and \(R_2\) are equal modulo renaming of variables and \(p = \epsilon\), then we call the overlap and critical pair \textbf{trivial}.

If \(R_1\) and \(R_2\) are freshened versions of the same rule and \(p = \epsilon\), then we call the overlap and critical pair \textbf{fresh root-permutative}.

A fresh overlap that is not trivial and not root-permutative is \textbf{proper}.

The fresh critical pair \(\Gamma \vdash (r_1\theta, l_1\theta[p \leftarrow r_2\theta])\) is \textbf{joinable} if there is a term \(u\) such that \(\Gamma \vdash r_1\theta \rightarrow_{c} u\) and \(\Gamma \vdash (l_1\theta[p \leftarrow r_2\theta]) \rightarrow_{c} u\).

Definition 5.2 Call a rewrite theory \(R = (\Sigma, Rw)\) \textbf{fresh orthogonal} (resp. \textbf{fresh quasi-orthogonal}) when all rules are left-linear and there are no non-trivial (resp. proper) fresh critical pairs.

Theorem 5.3 (Critical Pair Lemma for Closed Rewriting) Let \(R\) be a uniform rewrite theory where every non-trivial fresh critical pair is joinable. Then the closed rewriting relation generated by \(R\) is locally confluent.

Let \(R\) be a closed rewrite theory where every proper fresh critical pair is joinable. Then the closed rewriting relation generated by \(R\) is locally confluent.

Let \(R\) be a uniform rewrite theory where every proper fresh critical pair is joinable. Then the closed rewriting relation generated by \(R\) is locally confluent.

Proof By case analysis, considering the different kinds of peaks that may arise. The proofs for the statements are the same except in the last case.
• The rewrite steps defining the peak occur at disjoint positions. Then the peak is trivially joinable by applying the same rules and substitutions.

• The peak is generated by an overlap at a variable position. Consider \( R_1 = \Delta \vdash l_1^1 \rightarrow r_1^1 \) and \( R_2 = \Delta \vdash l_2^2 \rightarrow r_2^2 \) freshened versions of two rules (see Figure 2, but here we do not need permuted versions for the rules are already freshened). Let \( \Delta \) be the context used to rewrite \( l_2^2 \sigma_2 \) with \( R_1 \) and \( R_2 \). Without loss of generality, we assume \( \Delta, \text{atms}(R_1) \# \text{unkn}(\Delta, s) \vdash \Delta^1 \sigma_1, s \approx_\alpha l_1^1 \sigma_1, t \approx_\alpha r_1^1 \sigma_1 \) and \( s \) occurs inside \( l_2^2 \sigma_2 \) under an instance of an unknown \((\pi^e \cdot X^n) \sigma_2\). Then we can change the action of \( \sigma_2 \) over \( X^n \), replacing \( s \) by \( t \), as it is done in the first-order case. Here we rely on the assumption of uniformity, which ensures that no free atoms are introduced by the rewrite step, therefore no freshness constraint will be violated when replacing \( s \) by \( t \).

• There exist two freshened rules \( R_1 = \Delta_1 \vdash l_1^1 \rightarrow r_1^1 \) and \( R_2 = \Delta_2 \vdash l_2^2 \rightarrow r_2^2 \) and a term-in-context \( \Delta \vdash s \), such that there is a rewrite step at position \( p_1 \) in \( s \) using \( R_1 \) and at position \( p_2 \) using \( R_2 \). Then \( \Delta, \Gamma_1 \vdash \Delta_1 \sigma_1, l_1^1 \sigma_1 \approx_\alpha s | p_1 \) and \( \Delta, \Gamma_2 \vdash \Delta_2 \sigma_2, l_2^2 \sigma_2 \approx_\alpha s | p_2 \). Without loss of generality we assume that \( p_2 = p_1 q \). Since the sets of variables in the freshened rules are disjoint, without loss of generality we can assume \( \text{dom}(\sigma) \cap \text{dom}(\sigma') = \emptyset \), and define the substitution \( \mu = \sigma \cdot \sigma' \) such that \( \text{dom}(\mu) = \text{dom}(\sigma) \cup \text{dom}(\sigma') \). Then \( \Delta, \Gamma_1, \Gamma_2, \Delta_1 \mu, \Delta_2 \mu, l_1^1 q \mu \approx_\alpha l_2^2 q \mu \). Therefore the unification problem \( \Delta_1^1, \Delta_2, l_1^1 q \approx_\mu l_2^2 q \) has a solution. Hence, by Definition 5.1, there is a fresh critical pair between \( R_1 \) and \( R_2 \). Observe that, if \( R_1 = R_2 \) and \( q = \epsilon \) (trivial overlap), then the terms of divergence \( t_1 \) and \( t_2 \) are \( \alpha \)-equivalent. If the critical pair is non-trivial and the rules are uniform (part 1), it is joinable by assumption. If the theory is closed (part 2), we need to consider also a root-permutative overlap. In this case, joinability follows from \( \alpha \)-stability of closed rules (Lemma 4.9). If the rules are uniform (part 3), joinability follows using the fact that only rules without free atoms may have a root-permutative overlap (in which case, in the extended freshness context the two terms are \( \alpha \)-equivalent).

Therefore the peak is joinable since the rewriting relation is compatible with instantiation (the rules are uniform).

Since it is sufficient to consider just one freshened version of each rule when computing overlaps of closed rules, the number of fresh critical pairs for a rewrite theory with a finite number of rules is finite. Thus, Theorem 5.3 provides an effective criterion for local confluence, similar to the criterion for first-order systems.

We can deduce from Theorem 5.3 that the closed rewriting relation for the closed theory defining explicit substitution in Example 3.5 is locally confluent (every proper fresh critical pair is joinable). If we consider also the rule \((\text{Beta})\) then the system is not locally confluent. This does not contradict the previous theorem, because there is a non-proper fresh critical pair between \((\text{Beta})\) and \((\sigma \text{app})\) that is not joinable.

Next we consider criteria for confluence based on (quasi-) orthogonality.

Theorem 5.4 If \( R \) is a uniform and fresh-orthogonal rewrite theory, then the closed rewriting relation generated by \( R \) is confluent.

If \( R \) is a closed and fresh-quasi-orthogonal rewrite theory, then the closed rewriting relation generated by \( R \) is confluent.
If \( R \) is a uniform and fresh-quasi-orthogonal rewrite theory, then the closed rewriting relation generated by \( R \) is confluent.

**Proof** For each part the proof structure is the same as in [1,16,7], i.e., the diamond property is proved for a parallel closed-rewriting relation (simultaneous closed rewriting steps at disjoint positions). As the reflexive transitive closure of the parallel relation is equal to \( \Delta \vdash_R \rightarrow^*_c \Delta \), then the diamond-property of the parallel relation is sufficient to prove confluence of \( \Delta \vdash_R \rightarrow^*_c \Delta \). See Figure 3, and notice that if overlaps occur under instances of variables, we use the assumption of uniformity to ensure that when we change the substitution, the rewrite step is still possible. In Figure 3, the instances of variables are represented by pentagons. To prove the second and third parts of the theorem, we also need to show that root-permutative overlaps are joinable. This is a consequence of \( \alpha \)-stability in the case of closed rules; for uniform rules we use the fact that only rules without free atoms may have a root-permutative overlap.

**Example 5.5** Consider a signature for first-order logic, with term-formers \( \neg, \forall \) and \( \exists \) of arity 1, and \( \land, \lor \) of arity 2 (as usual we write them infix). The following closed rules can be used to simplify formulas.

\[
\begin{align*}
\vdash \neg (X \land Y) & \to \neg X \lor \neg Y \\
\vdash \neg (\forall [a]X) & \to \exists [b] \neg ((b \ a) \cdot X)
\end{align*}
\]

A nominal expert may be puzzled why we write \( \exists [b] \neg ((b \ a) \cdot X) \) on the right-hand side above, instead of the \( \alpha \)-equivalent \( \exists [a] \neg X \)? We could: these are equivalent—in a nominal context. The version given above is a direct translation of the corresponding CRS rule which, following Barendregt’s convention, must use different names for all bound variables in a term [6]. We can deduce from Theorem 5.4 that the closed rewriting relation generated by the theory in Example 5.5 is confluent. This theory is closed, but forbidden by ASP restrictions because of the permutation \( (b \ a) \) on the right-hand side.

The criteria for local confluence given in Theorem 5.3 and for confluence given in Theorem 5.4 for closed rewriting are easy to check using a nominal unification algorithm: we simply need to compute overlaps for the set of rules obtained by taking one freshened copy of each given rule. For comparison, the criteria given in both [7] and [19] require the computation of critical pairs for permutative variants of rules, which needs equivariant unification (NP).

Closed rules are uniform, but Theorems 5.3 and 5.4 apply to uniform theories, even if the rules are not closed, as long as we use closed rewriting. Consider the uniform rules \( \vdash f(a) \to 0 \) and \( \vdash g(f(b)) \to 0 \). These rules have no non-trivial fresh overlap, and closed rewriting is confluent, but the standard rewriting relation is not confluent, since the term \( g(f(a)) \) rewrites to both \( g(0) \) and 0. Using closed rewriting, the term \( g(f(a)) \) is a normal form.

### 6 Conclusion

We have presented easy-to-check criteria for confluence of nominal rewriting theories (Theorem 4.6 and Lemma 4.9, and Theorems 5.3 and 5.4), improving over the
criteria given in [7,19]. The Critical Pair Lemma for closed rewriting gives rise to a completion algorithm for closed rewrite rules (see [9]). In the future, we plan to enlarge the PVS library on term rewriting systems [10] including a formalisation of the results presented in this paper.

References

Proving Correctness of a Compiler Using Step-indexed Logical Relations

Leonardo Rodríguez\(^1\)  Miguel Pagano\(^2\)  Daniel Fridlender\(^3\)

FaMAF
Universidad Nacional de Córdoba
Argentina

Abstract

In this paper we prove the correctness of a compiler for a call-by-name language using step-indexed logical relations and biorthogonality. The source language is an extension of the simply typed lambda-calculus with recursion, and the target language is an extension of the Krivine abstract machine. We formalized the proof in the Coq proof assistant.

Keywords: Compiler verification, proof assistants, biorthogonality, step-indexed logical relations.

1 Introduction

There are many tools and frameworks available to analyze programs and to prove desirable properties about them, for instance, that they meet their specification. Several methods of static analysis such as program verification, and abstract interpretation can be used to lower the chance of letting errors go into deployed programs. However, a machine seldom executes source programs directly. Instead, they are translated into low-level programs with the help of a compiler. Therefore, we must consider the potential errors that the compilation process might introduce: a naive translation of a source program may easily invalidate its properties, making the effort initially invested useless. Dynamic program analysis, such as testing, may help finding errors in the executable code, but it is not enough when it comes to critical systems, which demand greater guarantees of security and reliability. It becomes necessary to prove that the compiler preserves semantics, that is, that the program generated by the compiler behaves exactly as the semantics of the source program indicates.

\(^1\) Email: lrodrig2@famaf.unc.edu.ar
\(^2\) Email: mpagano@famaf.unc.edu.ar
\(^3\) Email: dfridlender@famaf.unc.edu.ar

Selected, revised papers will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
Since the first proofs of compiler correctness appeared many years ago [20,21], there has been a considerable amount of progress in the topic. Of particular importance is the work of the CompCert project [17], a certified compiler for a large subset of the C programming language. In the case of functional languages we can mention [8] which is a certified compiler for the the simply-typed lambda calculus, and [4] where the source language is a call-by-value functional language, and the target is a variant of the SECD machine [16].

In order to prove a compiler correct it is necessary to find a connection between the semantics of the source language and the semantics of the target language. In general, the latter is described operationally: we define which are the instructions available in the machine (a real microprocessor or an abstract machine) and how those instructions modify the configuration as they are executed. On the other hand, there are many ways to describe the semantics of the source language, and the structure of the proof of correctness is highly dependent on which method is used. There are proofs of compiler correctness based on the big-step semantics [9,19], small-step semantics [2], or denotational semantics [4,8] of the source language, among others.

In this work we prove the correctness of a compiler for a typed call-by-name functional language, and the proof is based on the domain-theoretic denotational semantics of the language. The compiler translates a well-typed term of the source language into a list of instructions for the Krivine machine (KAM) [15]. We use step-indexed logical relations [1,3] and biorthogonality [23] to capture the notion of correctness in a compositional and modular way. These two techniques have been used before in combination to obtain proofs of compiler correctness [4,5,12] and also applied in other topics such as program equivalence [11]. As far as we know, no previous work has applied these techniques to prove the correctness of a compiler targeting the KAM and for a call-by-name language.

The approach we follow in this paper has been used before by [4] but applied to a call-by-value language and the SECD machine. In this work we revise the method in such a way that it becomes applicable in a call-by-name language and the KAM machine, and we obtained a simpler definition of the logical relations and a cleaner proof of correctness.

We formalized all the results in the Coq proof assistant, and the code is available online [25]. We used and extended a domain theory library [6] as a basis for the formalization of the semantics and the logical relations.

The rest of the paper is organized as follows. In Section 2 we present the source language and its denotational semantics. We continue in Section 3 with the target language and its operational semantics. We present a general explanation of biorthogonality in Section 4 and then we apply this technique in Section 5 in which we present our first logical relation that we called “denotational approximation”. In Section 6 we introduce step-indexing and some results about its combination with biorthogonality. We apply both biorthogonality and step-indexing in Section 7 to construct the second logical relation called “operational approximation”. We comment on the formalization in Coq in Section 8 and in Section 9 we conclude.
2 The Source Language

The terms of the source language are the following:

Definition 2.1 (Language terms).

\[ \mathcal{T} \ni t ::= \lambda t \mid t_1 t_2 \mid n \mid \text{rec } t \mid m \mid \ominus^n (t_1, \ldots, t_n) \]
\[ \mid (t_0, t_1) \mid \text{fst } t \mid \text{snd } t \mid \text{ifz } t . t' \]

Hereafter we use the notation \( \mathcal{T} \ni t \) to specify both the set defined by the grammar and our naming convention for meta-variables ranging over it. The first three constructors correspond to the lambda calculus with de Bruijn indices. The language also includes a fixed-point operator, integer constants, strict arithmetic operators, pairs and projections. The last constructor is a conditional projection. We choose this form of conditional for convenience, but a more familiar constructor of the form \( \text{ifz } t \text{ then } t_1 \text{ else } t_2 \) can be expressed as \( \text{ifz } t . (t_1, t_2) \). We write \( \ominus^n \) to represent any strict arithmetic operator with arity \( n > 0 \); operators are written in prefix position and cannot be partially applied.

The type system is rather simple. We have a single basic type \( \text{int} \), and also arrow and product types. A context is defined to be a list of types, accordingly with the use of de Bruijn indices.

Definition 2.2 (Types and contexts).

\[ \Theta \ni \theta ::= \text{int} \mid \theta \rightarrow \theta' \mid \theta \times \theta' \]
\[ \Theta^* \ni \pi ::= [] \mid \theta :: \pi \]

We present the typing rules for the language in Figure 1, which are quite familiar. The conclusion of a typing rule is a judgement of the form \( \pi \vdash t : \theta \) which states that the term \( t \) has type \( \theta \) under the context \( \pi \).
2.1 Denotational Semantics

The denotational semantics of the source language is given in a domain-theoretic setting because of the presence of the fixed-point operator. In this section, and in the rest of the paper, we will follow a traditional treatment of domain theory – for example, we will not comment on how one calculates the supremum of a chain. In contrast, our formalization in Coq is based on a constructive domain theory library [6] where the supremum is given by a function (in Coq’s language).

Before coming to the semantics of the language, we recall some concepts and notations of domain theory. The domain of continuous functions from a domain \( P \) to a domain \( Q \) is written as \([P \rightarrow Q]\). Any set \( P \) can be turned into the flat domain \( P_\perp \) by adjoining a least element \( \perp \); this construction can be turned into a Kleisli triple whose unit is \( \pi_\perp : P \rightarrow P_\perp \) and its extension operation for a \( f : P \rightarrow Q_\perp \) is given by \( f^\perp \perp = \perp \) and \( f^\perp(x) = f x \), for \( x \in P \). The semantics of strict arithmetic operators are based on abstract considerations: for a total function \( \oplus : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \) we can get a function \( \oplus^\ast : (\mathbb{Z} \times \mathbb{Z})_\perp \rightarrow \mathbb{Z}_\perp \) by \( \oplus^\ast = (\pi_\perp \oplus)_\perp^\ast \). Since the denotation of operands will be an unlifted tuple of lifted values, we compose \( \oplus^\ast \) with the strength, \( \tau_{A,B} : A_\perp \times B_\perp \rightarrow (A \times B)_\perp \), to get the function \( \oplus_\perp = \oplus^\ast \cdot \tau_{\mathbb{Z},\mathbb{Z}} \); notice that one can apply the same construction for an \( n \)-ary operation. Given a function \( f : P \rightarrow D \) from the predomain \( P \) to a domain \( D \), we write \( f_\perp : P_\perp \rightarrow D \) for the function such that \( f_\perp \perp = \perp \) and \( f_\perp(x) = f x \), for \( x \in P \). If \( f : D \rightarrow D \) is a continuous function over the domain \( D \), then \( Y_D f \) denotes the least fixed-point of \( f \).

As usual, once we choose a domain for the denotation of atomic types, in our case \textbf{int}, the semantics of arrow types and contexts are determined by the exponentials and finite products of the underlying category.

\begin{definition} \textbf{(Semantics of types and contexts).} \end{definition}

\[
\begin{align*}
\sem{\textbf{int}} &= \mathbb{Z}_\perp \\
\sem{\theta \rightarrow \theta'} &= \sem{\theta} \rightarrow \sem{\theta'} \\
\sem{\theta \times \theta'} &= \sem{\theta} \times \sem{\theta'} \\
\sem{\theta : \pi} &= \sem{\theta} \times \sem{\pi}
\end{align*}
\]

In Figure 2 we present the denotational semantics for typing derivations of the source language. We use the symbol \( \lambda \) as a meta-binder to avoid confusion with the symbol \( \lambda \) used in abstractions. Also, if \( \gamma \) is a finite product, we write \( \gamma \downarrow n \) for its \( n \)-th projection. The \textit{coherence} of this semantics – meaning that different judgements of the same expression have the same denotation – has already been proved for a language even larger than the one we use in this paper [24].

\textit{Semantic Chain} \smallskip

The semantics of a typing judgement can be thought of as the limit of increasingly better defined denotational values. In Figure 3 we define a family of functions \( \sem{-} \), indexed by natural numbers. It is easy to see that \( \sem{\pi \vdash t : \theta} \subseteq \sem{\pi \vdash t : \theta}_{i+1} \), thus the sequence \( \sem{\pi \vdash t : \theta}_i \) forms a chain in the domain \( \sem{\pi} \rightarrow \sem{\theta} \) and its supremum is \( \sem{\pi \vdash t : \theta} \).

Later we prove that each element of the semantic chain \( \sem{\pi \vdash t : \theta}_i \), approximates the compiled code for \( t \), a key point in the correctness proof.
\[
\begin{align*}
\llbracket \_ \rrbracket & : \pi \vdash t : \theta \rightarrow \llbracket \_ \rrbracket \rightarrow \llbracket \theta \rrbracket \\
\llbracket \pi \vdash \lambda t : \theta \rightarrow \theta' \rrbracket \gamma &= \lambda a. \llbracket \theta :: \pi \vdash t : \theta' \rrbracket (a, \gamma) \\
\llbracket \pi \vdash t_1 t_2 : \theta' \rrbracket \gamma &= \left( \llbracket \pi \vdash t_1 : \theta \rightarrow \theta' \rrbracket \gamma \right) \left( \llbracket \pi \vdash t_2 : \theta \rrbracket \gamma \right) \\
\llbracket \pi \vdash \pi . n \rrbracket \gamma &= \gamma \downarrow n \\
\llbracket \pi \vdash rec t : \theta \rrbracket \gamma &= Y_{\llbracket \theta \rrbracket} \left( \llbracket \pi \vdash t : \theta \rightarrow \theta \rrbracket \gamma \right) \\
\llbracket \pi \vdash m : \mathbb{int} \rrbracket \gamma &= \iota \uparrow m \\
\llbracket \pi \vdash \ominus^n (t_1, \ldots, t_n) : \mathbb{int} \rrbracket \gamma &= \ominus^n (d_1, \ldots, d_n) \\
& \text{where } d_j = \llbracket \pi \vdash t_j : \mathbb{int} \rrbracket \gamma \\
\llbracket \pi \vdash \text{fst } t : \theta \rrbracket \gamma &= d_0 \\
& \text{where } (d_0, d_1) = \llbracket \pi \vdash t : \theta \times \theta' \rrbracket \gamma \\
\llbracket \pi \vdash \text{snd } t : \theta \rrbracket \gamma &= d_1 \\
& \text{where } (d_0, d_1) = \llbracket \pi \vdash t : \theta \times \theta' \rrbracket \gamma \\
\llbracket \pi \vdash (t_0, t_1) : \theta \times \theta' \rrbracket \gamma &= \left( \llbracket \pi \vdash t_0 : \theta \rrbracket \gamma, \llbracket \pi \vdash t_1 : \theta' \rrbracket \gamma \right) \\
\llbracket \pi \vdash ifz t . t' : \theta \rrbracket \gamma &= (\lambda z . \text{if } z = 0 \text{ then } d_0 \text{ else } d_1) \downarrow d \\
& \text{where } d = \llbracket \pi \vdash t : \mathbb{int} \rrbracket \gamma \text{ and } (d_0, d_1) = \llbracket \pi \vdash t' : \theta \times \theta \rrbracket \gamma
\end{align*}
\]

Fig. 2: Denotational semantics.

3 The Target Language

3.1 The Abstract Machine

We use an abstract machine as our target language. Abstract machines are often used as an idealized model of execution; they are in general simpler than a real machine since they lack certain hardware details that would otherwise complicate the reasoning and the analysis of its behaviour. They are therefore suitable as an intermediate language for compilation [10]. We proceed with the definition of the components of our machine. The instructions are the following:

**Definition 3.1** (Machine instructions).

\[
I \ni c, c' ::= \text{Grab } c \mid \text{Push } c \triangleright c' \mid \text{Access } n \\
& \mid \text{Fix } c \triangleright \text{Pair } (c, c') \mid \text{Fst} \mid \text{Snd} \\
& \mid \text{Frame } \ominus^n \mid \text{Op} \mid \text{Const } m
\]

The first three instructions correspond to the classic KAM that are sufficient to evaluate the pure lambda calculus. We have added new instructions to handle recursion, strict arithmetic operators, pairs, and conditionals. We now define the
components of the machine and some meta-variables as follows:

**Definition 3.2** (Machine components).

Closures: \[ \Gamma \ni \alpha := (c, \eta) \]

Environments: \[ H \ni \eta := [] \mid \alpha :: \eta \]

Operators: \[ Ops \ni \ominus^n \]

Stack values: \[ M \ni \mu := \alpha \mid [\ominus^n \overline{m} \bullet \overline{a}] \mid \langle \alpha, \alpha' \rangle \]

Stacks: \[ S \ni s := [] \mid \mu :: s \]

Configurations: \[ W = \Gamma \times S \ni w := (\alpha, s) \]

A machine configuration is a pair \((\alpha, s)\) where \(\alpha\) is a closure and \(s\) is a stack. A closure is also a pair \((c, \eta)\) where \(c\) is an instruction and \(\eta\) is a machine environment; which is itself a list of closures. A stack value can be either a closure, a frame, or a pair of closures. We use frames [27] to store the arguments of operators throughout execution: a frame \([\ominus^n \overline{m} \bullet \overline{a}]\) has three components: the list \(\overline{m}\) of arguments already computed, a hole to indicate the argument that is being computed at the time, and a list \(\overline{a}\) of closures for computing the remaining arguments. The transition
rules of the KAM are shown in Figure 4; they define a deterministic relation \( \rightarrow \subseteq W \times W \). We use the symbol "|" in the rules to help the reader to distinguish the components of the configuration.

### 3.2 Compilation

Now that we have defined both the source and the target language, we present in Figure 5 the compilation function: each well-typed term is mapped into KAM’s code. For a closed term \( t \) of type \( \text{int} \), we expect that the execution of \( (\langle t \rangle, \text{int}, []) \) leads to a configuration \( ((\text{Const } m, \eta), []) \) if \( \llbracket t \rrbracket(\cdot) = \nu \uparrow m \). In the next sections we will prove that statement.

### 4 Biorthogonality

Biorthogonality is a well-known technique that has been used in program equivalence [23], realizability [14], and compiler correctness [4,12]. The general idea can be explained as follows.

Let \( \mathcal{E} \) and \( \mathcal{T} \) be two sets, and \( \models \subseteq \mathcal{E} \times \mathcal{T} \) a relation between those two sets. If
we think $T$ as a set of tests, and $\models$ as a satisfiability relation, then $e \models t$ states whether an element $e \in E$ satisfies the test $t \in T$. Suppose $T_0$ is a subset of $T$, we write $T_0^\top$ for the set of elements that satisfy all the tests in $T_0$:  

$$T_0^\top = \{ e \in E \mid \text{for all } t \in T_0, e \models t \}.$$ 

As a concrete example, if $T$ are formulas and $E$ are models of a particular logic, then $T_0^\top$ is the set of models that satisfy all the formulas in $T_0$. We can also define a dual operation to obtain the set of tests that are satisfied by all the elements in a subset $E_0 \subseteq E$:  

$$E_0^\perp = \{ t \in T \mid \text{for all } e \in E_0, e \models t \}.$$ 

The operators $\perp$ and $\top$ are often called \textit{orthogonal}, and it is a well-known fact that they form an antitone Galois connection $[7,22]$. As a consequence, the function $(\cdot)^\perp : \mathcal{P}(E) \to \mathcal{P}(E)$ is a closure operator for the poset $(\mathcal{P}(E), \subseteq)$.

The key point of biorthogonality is that for a given set $E_0$ we can obtain the set $E_0^{\perp \top}$ which is an extension of $E_0$ that satisfies all the tests in $E_0^\top$. That is, we are able to extend the set $E_0$ without “losing” any test and hence maintaining the satisfiability relation. In the next section we present the concrete use of biorthogonality that is useful for our purposes.

### 5 Denotational Approximation

In this section we prove the correctness of the compiler for terms whose denotation is different from bottom. The strategy is to define a logical relation which states when a denotational value $d$ approximates a closure $\alpha$ at type $\theta$; then we prove the fundamental lemma of logical relations, finally concluding that the compilation of a term is approximated by every element of its semantic chain.
Our logical relation is parameterized over a set of observations of the KAM. Given a set of observations \( R \subseteq W = \Gamma \times S \), we use biorthogonality to define this logical relation – following the terminology of the previous section, we will say that stacks are tests for closures. All the reasoning of this section assumes that \( R \) is closed by anti-execution, i.e. \((\alpha, s) \in R \) and \((\alpha', s') \rightarrow^{*} (\alpha, s)\) implies \((\alpha', s') \in R\); moreover, to keep our reasoning constructive we will also ask for the existence of an “excluded” closure: that is an \( \hat{\alpha} \) such that \((\hat{\alpha}, s) \not\in R\) for any \( s \in S \) (a closure that does not satisfy any test). Termination is an example of an observation which is closed by anti-execution and has an excluded closure (think of the compilation of \( \text{rec} \; \lambda x.x\)).

The relations \( \triangleright^{\theta}, \blacktriangleright^{\theta} \subseteq \Gamma \times \llbracket \theta \rrbracket \) are defined by mutual recursion over types as follows:

**Definition 5.1** (Denotational approximation).

\[
\alpha \triangleright^{\theta} \bot \text{ for any closure } \alpha,
\]

\[
(\text{Const } m, \eta) \triangleright^{\text{int}} \uparrow m,
\]

\[
(\text{Grab } c, \eta) \triangleright^{\theta \rightarrow \theta'} f \text{ iff for all } \alpha \text{ and } d, \text{ if } \alpha \blacktriangleright^{\theta} \ d \text{ then } (c, \alpha :: \eta) \blacktriangleright^{\theta'} f \ d,
\]

\[
(\text{Pair } (c_0, c_1), \eta) \triangleright^{\theta \times \theta'} (d_0, d_1) \text{ iff } (c_0, \eta) \blacktriangleright^{\theta} d_0 \text{ and } (c_1, \eta) \blacktriangleright^{\theta'} d_1,
\]

\[
\alpha \blacktriangleright^{\theta} d \text{ iff } \alpha \in \Gamma^{\theta}(d)^{\perp_{R}}, \text{ where } \Gamma^{\theta}(d) = \{ \alpha \mid \alpha \triangleright^{\theta} d \}.
\]

Let \( \alpha \in \Gamma \) and \( d \in \llbracket \theta \rrbracket \), then \( \alpha \blacktriangleright^{\theta} d \) is read “\( d \) is an approximation of type \( \theta \) to the closure \( \alpha \)” but we often omit the type and just write “\( d \) approximates the closure \( \alpha \)”.

In a sense, the set \( \Gamma^{\theta}(d) \) contains the closures that are “strongly approximated” by \( d \), and \( \Gamma^{\theta}(d)^{\perp_{R}} \) is the extension of this set obtained through the orthogonal operators. Note that, in this definition, the transitions of the machine are not relevant except in the restrictions we imposed to the set of observations.

We let \( \bot \) to be an approximation of any closure; this is consistent with the idea of approximation since \( \bot \) is a value with a minimum amount of information. Since \( \Gamma^{\theta}(\bot) = \Gamma \), by the “excluded closure” assumption, we know \( \Gamma^{\theta}(\bot)^{\perp_{R}} = \emptyset \); consequently \( s \in \Gamma^{\theta}(d)^{\perp_{R}} \) always implies \( d \neq \bot \). The fact that \( R \) is closed by anti-execution leads to the following (trivial) lemma:

**Lemma 5.2** Let \( \alpha, \alpha' \in \Gamma \) and \( s, s' \in S \). If \( (\alpha', s') \rightarrow^{*} (\alpha, s) \), \( \alpha \blacktriangleright^{\theta} d \) and \( s \in \Gamma^{\theta}(d)^{\perp_{R}} \) then \( (\alpha', s') \in R \).

We define a relation \( \blacktriangleright^{\pi} \) between machine-level environments and denotational environments, as a point-wise extension of the \( \blacktriangleright^{\theta} \) relation.

**Definition 5.3** (Denotational approximation for environments).

\[
\alpha :: \eta \blacktriangleright^{\theta :: \pi} (d, \gamma) \text{ iff } \alpha \blacktriangleright^{\theta} d \text{ and } \eta \blacktriangleright^{\pi} \gamma.
\]

If we follow the general schema of biorthogonality presented before, the elements of \( \Gamma^{\theta}(d)^{\perp_{R}} \) are the tests that a closure must satisfy to be approximated by the value \( d \). Since this set depends on the type \( \theta \), we can also talk about “tests of type \( \theta \)”.
(which is a frequent terminology in the literature about realizability). In Krivine’s realizability tests of arrow types $\theta \to \theta'$ are stacks $\alpha :: s$ where $\alpha$ is a realizer of $\theta$ and $s$ is a test of type $\theta'$; as the following lemma shows, that is a good characterization in our setting.

**Lemma 5.4** Let $\alpha \in \Gamma$, $s \in S$, $f \in \llbracket \theta \to \theta' \rrbracket$, $d \in \llbracket \theta \rrbracket$. If $\alpha \toplus \theta d$ and $s \in \Gamma^{\theta}(f d)^{1-R}$ then $\alpha :: s \in \Gamma^{\theta}^{\theta'}(f)^{1-R}$.

**Proof.** In order to prove $\alpha :: s \in \Gamma^{\theta}^{\theta'}(f)^{1-R}$ we take $\alpha' \in \Gamma^{\theta}^{\theta'}(f)$ and prove $(\alpha', \alpha :: s) \in R$. Since $\Gamma^{\theta}^{\theta'}(f d)^{1-R}$ is not empty we know $f d \neq \bot$ and hence $f \neq \bot$. Therefore, we have by inversion that $\alpha'$ is a closure of the form $(\text{Grab } c, \eta)$ where $\eta \in H$ and $c \in I$. Moreover, since $\alpha \toplus \theta d$ we have $(c, \alpha :: \eta) \toplus \theta' f d$. Consequently, since $(\alpha', \alpha :: s) \mapsto ((c, \alpha :: \eta), s)$ and $s \in \Gamma^{\theta}^{\theta'}(f d)^{1-R}$ we conclude $(\alpha', \alpha :: s) \in R$. □

Analogously, it is easy to see that tests for a product type $\theta \times \theta'$ can be defined as those stacks having a “projection” at the top followed by a test for the projected type. For the sake of brevity we do not show characterization of tests for $\text{int}$, which can be found in the formalization.

**Lemma 5.5** Let $s \in S$, $\eta \in H$, $d_0 \in \llbracket \theta \rrbracket$, $d_1 \in \llbracket \theta' \rrbracket$. If $s \in \Gamma(\theta)(d_0)^{1-R}$ then $(\text{Fst, } \eta) :: s \in \Gamma^{\theta \times \theta'}(\llbracket (d_0, d_1) \rrbracket)^{1-R}$. Similarly, if $s \in \Gamma^{\theta}(d_1)^{1-R}$ then $(\text{Snd, } \eta) :: s \in \Gamma^{\theta \times \theta'}(\llbracket (d_0, d_1) \rrbracket)^{1-R}$.

The next lemma provides various ways to combine closures using machine instructions, in order to obtain new approximations of different types. This is an important property since it essentially says that we can merge “correct” code fragments (potentially generated by different compilers, or hand-written) to obtain a larger code fragment that is also correct.

**Lemma 5.6** (i) If $(c, \eta) \toplus \theta f$ and $(c', \eta) \toplus \theta d$, then $(\text{Push } c' \triangleright c, \eta) \toplus \theta f d$.
(ii) If $\eta \toplus \pi \gamma$ and $n < |\pi|$, then $(\text{Access } n, \eta) \oplus \pi n \gamma \perp n$.
(iii) If $(c, \eta) \toplus \theta f$ and $(\text{Fix } c, \eta) \oplus \theta d$, then $(\text{Fix } c, \eta) \oplus \theta f d$.
(iv) If $(c_i, \eta) \oplus \text{int } d_i$ for all $i \in \{1, \ldots, n\}$, then $(\text{Push } c_n \triangleright \ldots \triangleright \text{Push } c_1 \triangleright \text{Frame } \ominus \eta) \oplus \text{int } \ominus \eta (d_1, \ldots, d_n)$.
(v) If $(c, \eta) \oplus \theta \times \theta' (d_0, d_1)$, then $(\text{Push } \text{Fst } c, \eta) \oplus \theta d_0$ and $(\text{Push } \text{Snd } c, \eta) \oplus \theta' d_1$.
(vi) If $(c, \eta) \oplus \theta \times \theta' (d_0, d_1)$ and $(c', \eta) \oplus \text{int } d$, then $(\text{Push } c' \triangleright c, \eta) \oplus \theta (\lambda z. \text{if } z = 0 \text{ then } d_0 \text{ else } d_1) \equiv d$.

By using Lemma 5.6, we can easily prove that the compilation of a typing derivation is related with every element of its semantic chain.

**Lemma 5.7** (Denotational approximation of compiled code) If $\eta \oplus \pi \gamma$ then for all $i$, $(\llbracket t \rrbracket_{\pi, \theta}, \eta) \oplus \theta \llbracket \pi \vdash t : \theta \rrbracket_i \gamma$.

**Proof.** The proof is by induction in the typing derivation. However, in the case of the fixed-point operator, we need a nested induction over the index $i$. We now show the proof for that case.

Let $c = \llbracket t \rrbracket_{\pi, \theta}$, let $f_i = \llbracket \pi \vdash t : \theta \rrbracket_i \gamma$, and $d_i = \llbracket \pi \vdash \text{rec } t : \theta \rrbracket_i \gamma$. We want to prove $(\text{Fix } c, \eta) \oplus \theta d_i$ by induction over $i$. The case $i = 0$ is trivial.
since \( d_0 = \perp \) (and \( \perp \) is always an approximation). In the inductive case, we assume 
\( (\text{Fix} \triangleright c, \eta) \triangleright^\theta d_i \) and prove \( (\text{Fix} \triangleright c, \eta) \triangleright^\theta d_{i+1} \). We have \((c, \eta) \triangleright^\theta \triangleright^\theta f_i\) by inductive hypothesis, and hence by Lem. 5.6 we get \((\text{Fix} \triangleright c, \eta) \triangleright^\theta \triangleright^\theta f_i d_i = d_{i+1}. \Box\)

It is possible to relate the compilation of a term directly to its semantics by defining an admissible extension of \( \triangleright^\theta \); the interested reader is invited to consult this extension in the formalization.

Note that Lemma 5.7 holds for any choice of \( R \) that satisfies the two conditions we stated before: it must be closed by anti-execution and there must be an excluded closure. In particular, to prove a “standard” version of the compiler correctness theorem for closed terms of type \textbf{int} one fixes the set of observation to be \( R_m = \{ w \in W \mid w \rightarrow^* ((\text{Const} m, \eta), []) \} \) for any environment \( \eta \).

**Lemma 5.8** If \( t \) is a closed term, and \( \llbracket \llbracket t \rrbracket : \text{int} \rrbracket \vdash t : \text{int} \rrbracket_i () \) = \( \iota^m \), then there is some environment \( \eta \) such that \( (\llbracket t \rrbracket : \text{int} \rrbracket, \eta), []) \rightarrow^* ((\text{Const} m, \eta), []). \)

**Proof.** In order to prove this result, we use Lemma 5.7 choosing \( R_m \) as the set of observations. We have then \( (\llbracket t \rrbracket : \text{int} \rrbracket, \eta), []) \triangleright^\text{int} \llbracket \llbracket t \rrbracket : \text{int} \rrbracket_i () \) for all \( i \in \mathbb{N} \). But since \( \llbracket \text{int} \rrbracket \) is a flat domain, there is a \( j \in \mathbb{N} \) such that \( \llbracket \llbracket t \rrbracket : \text{int} \rrbracket_j () = \iota^m \) and hence we have \( (\llbracket t \rrbracket : \text{int} \rrbracket, \eta), []) \triangleright^\text{int} \iota^m. \) Since \( \llbracket \eta \rrbracket \in \Gamma^{\text{int}(\iota^m)^{-n_m}} \), by Lemma 5.2 we have \((\llbracket t \rrbracket : \text{int} \rrbracket, \eta), []\rrbracket \in R_m \), which is what we wanted by the definition of \( R_m. \Box \)

In order to prove a similar lemma for divergent terms we use another logical relation with similar properties.

## 6 Step-indexed Logical Relations

To prove the correctness of the compiler for terminating terms it was necessary to relate code fragments with each element of the semantic chain; as the proof of Lemma 5.7 shows, this allowed us to make a nested induction when considering the case for \( \text{rec} \ t \). If the source language were strong normalizing (i.e., by setting aside the fixed point operator), there would be no need to introduce the semantic chain and correctness would relate the compiled code with the semantics of the term.

A more general approach to deal with the subtleties introduced by the recursion operator is using \textit{step-indexed} logical relations. This method has been used alone and in combination with biorthogonality to obtain proofs of compiler correctness [5,12] and program equivalence [1,11], among other topics. The basic idea is that the logical relation is defined incrementally through a \textit{family} of relations indexed by natural numbers. Thus, one can prove different properties about this relation using induction in the index. Step-indexing is helpful to capture a notion of approximation at the operational side, analogous to that provided by the semantic chain at the denotational side. In this section we introduce step-indexed families and show some results regarding the combination of these families with the orthogonal operators.

**Definition 6.1** (Step-indexed family). A family \( R_i \subseteq A \) is step-indexed if \( R_0 = A \) and for all \( i \in \mathbb{N}, R_{i+1} \subseteq R_i. \)

An example of a step-indexed family over the set of KAM configurations is given by letting \( R_i \) be the set of configurations that can make at least \( i \) transition steps.
While in the previous section we parameterized all the development over a set of observations, in the next section we will work with any step-indexed family of observations closed by anti-execution.

Given a family of observations $R_i \subseteq \mathcal{E} \times \mathcal{T}$, we can define a binary relation $R \subseteq \hat{\mathcal{E}} \times \hat{\mathcal{T}}$ over indexed elements $\hat{\mathcal{E}} = \mathbb{N} \times \mathcal{E}$ and indexed tests $\hat{\mathcal{T}} = \mathbb{N} \times \mathcal{T}$.

**Definition 6.2** Let $R_i \subseteq \mathcal{E} \times \mathcal{T}$ be an indexed family, then $R \subseteq \hat{\mathcal{E}} \times \hat{\mathcal{T}}$ is given by $(i,e) R (j,t)$ iff $(e,t) \in R_{\min(i,j)}$.

Let us make explicit the definition of the orthogonal operator $(\_)^{\perp_R}$ for the relation $R$:

$$X^{\perp_R} = \{ (j,t) \in \hat{T} \mid \text{for all } (i,e) \in X, (i,e) R (j,t) \} ,$$

which means that to prove $(j,t) \in X^{\perp_R}$ one has to check that every element $(i,e)$ in $X$ is related with $(j,t)$ via $R$. Now we prove that one can simplify the reasoning when $R_i$ is step-indexed.

**Definition 6.3** (Down-closed set). For any set $\mathcal{E}$, we say that $X \subseteq \hat{\mathcal{E}}$ is down-closed if whenever $(i,e) \in X$ and $j \leq i$, then $(j,e) \in X$.

**Lemma 6.4** Let $R_i \subseteq \mathcal{E} \times \mathcal{T}$ be step-indexed and $X \subseteq \hat{\mathcal{E}}$, then $X^{\perp_R}$ is down-closed.

**Proof.** Let $(j,t) \in X^{\perp_R}$ and $i \leq j$. Suppose $(k,e) \in X$, then $(e,t) \in R_{\min(k,j)}$. Since $\min(k,i) \leq \min(k,j)$, we have $R_{\min(k,j)} \subseteq R_{\min(k,i)}$ and hence $(e,t) \in R_{\min(k,i)}$. Therefore, $(i,e) \in X^{\perp_R}$. $\square$

As the following lemma shows, when restricted to down-closed sets, one can give a simpler definition of $(\_)^{\perp_R}$ in which it is only necessary to check those elements that satisfy $i \leq j$.

**Lemma 6.5** Let $R_i \subseteq \mathcal{E} \times \mathcal{T}$. If $X$ is down-closed, then

$$X^{\perp_R} = \{ (j,t) \in \hat{T} \mid \text{for all } (i,e) \in X, i \leq j \text{ implies } (e,t) \in R_i \} .$$

Since Lemmas 6.4 and 6.5 can also be proved for the operator $(\_)^{\top_R}$, we can construct an alternative definition of the closure operator.

**Lemma 6.6** Let $R_i \subseteq \mathcal{E} \times \mathcal{T}$ be step-indexed and $X \subseteq \hat{\mathcal{E}}$, then

$$X^{\perp_R \top_R} = \{ (j,e) \in \hat{\mathcal{E}} \mid \text{for all } (i,t) \in X^{\perp_R}, i \leq j \text{ implies } (e,t) \in R_i \} .$$

### 7 Operational Approximation

In this section we define a relation of approximation from machine closures to denotational values. In the same vein as in section 5, these relations are defined by means of the closure operator associated to a set of observations. The logical relation is parameterized by a step-indexed family of relations $R_i \subseteq \Gamma \times S$ satisfying the following condition: $(\alpha, s) \in R_i$ and $(\alpha', s') \rightarrow (\alpha, s)$ imply $(\alpha', s') \in R_{i+1}$. Note that this condition implies that each relation of the family is closed by anti-execution.

This time, instead of working with a single relation of approximation, we define simultaneously two families of relations, $\llbracket \theta \rrbracket, \llbracket \theta \rrbracket' \subseteq \Gamma \times \llbracket \theta \rrbracket$, indexed by natural numbers. Roughly speaking, the index measures the “accuracy” of the approximation:
the relation becomes finer as the index increases, starting with the total relation at index 0, where every closure approximates every denotation.

**Definition 7.1** (Operational approximation).

\[ \alpha \ll^\theta \ll^\Delta d, \]
\[ (\text{Const} \ m, \eta) \ll^\text{int} \ll^\Delta m, \]
\[ (\text{Grab} \ c, \eta) \ll^{\theta \Rightarrow \theta'} f \text{ iff } \forall k \leq i, \text{ if } \alpha \ll^\Delta k d \text{ then } (c, \alpha :: \eta) \ll^\Delta '\ll^\Delta k f d, \]
\[ (\text{Pair} \ (c_0, c_1), \eta) \ll^{\theta \times \theta'} (d_0, d_1) \text{ iff } (c_0, \eta) \ll^\Delta i d_0 \text{ and } (c_1, \eta) \ll^\Delta j d_1, \]
\[ \alpha \ll^\Delta k d \text{ iff } (k, \alpha) \in \Gamma^\theta(d)^{\ll^\Delta R}, \text{ where } \Gamma^\theta(d) = \{ (k, \alpha) \mid \alpha \ll^\theta d \}. \]

While in the denotational approximation (Def. 5.1), \( \bot \) was strongly related with any closure, now \( \bot \) is strongly approximated by any closure only at level 0. As a consequence, we have \((k + 1, \alpha) \not\in \Gamma^\text{int}(\bot)\); from this and from the fact that \( R_i \) is step-indexed, is easy to show \( \Gamma^\text{int}(\bot)^{\ll^\Delta R} = N \times S \). This implies that a closure \( \alpha \) which approximates \( \bot \) at every level must be a divergent closure.

**Lemma 7.2** Define \( R_i = \{ w \mid w \text{ can make at least } i \text{ transition steps} \} \). Let \( \alpha \in \Gamma \) such that \( \alpha \ll^\text{int} \bot \) for all \( k \in \mathbb{N} \), then \( (\alpha, s) \) diverges for any stack \( s \).

**Proof.** Given that \( \Gamma^\text{int}(\bot)^{\ll^\Delta R} = N \times S \), we have that for any pair \( (N, s) \in N \times S \) it holds \( (\alpha, s) \in R_N \). That is, \( (\alpha, s) \) can make an arbitrarily large number of transition steps.

The intuitive interpretation of the indices given above suggests that when \( \alpha \) is an approximation at index \( k \), it should also be an approximation at a smaller index \( j \leq k \). In addition, the approximation relation is monotone with respect to the domain order.

**Lemma 7.3** Let \( \alpha \in \Gamma \), \( d \subseteq d' \subseteq [\theta] \), and \( j \leq k \). If \( \alpha \ll^\Delta d \) then \( \alpha \ll^\Delta j d' \). Similarly, if \( \alpha \ll^\Delta k d \) then \( \alpha \ll^\Delta j d' \).

From this lemma we deduce that \( \Gamma^\theta(d) \) is down-closed and monotone: \( d \subseteq d' \) implies \( \Gamma^\theta(d) \subseteq \Gamma^\theta(d') \). Moreover the family \( \theta_k(\alpha) = \{ d \mid \alpha \ll^\Delta k d \} \) is step-indexed over \( [\theta] \), and each \( \theta_k(\alpha) \) is closed by suprema of chains.

It is not surprising that we can construct tests for compound types by combining tests for simpler types. Recall that this time “tests” are pairs \((k, s)\) where \( k \in \mathbb{N} \) and \( s \in S \). We only show here one way to obtain tests for product types, and there are other possible combinations that can be examined in the formalization.

**Lemma 7.4** Let \( \alpha \in \Gamma \), \( s \in S \). If \( \alpha \ll^\text{int} 0 \) and \( (k, s) \in \Gamma^\theta(d_0)^{\ll^\Delta R} \) then it holds \( (k, \alpha :: s) \in \Gamma^\theta \times \theta'((d_0, d_1))^{\ll^\Delta R} \). Similarly, if \( \alpha \ll^\text{int} \ll^\Delta m \) with \( m \neq 0 \) and \( (k, s) \in \Gamma^\theta(d_1)^{\ll^\Delta R} \) then \( (k, \alpha :: s) \in \Gamma^\theta \times \theta'((d_0, d_1))^{\ll^\Delta R} \).

The following lemma presents the operational counterpart of Lemma 5.6, showing that approximations compose well with the constructors of the language. We show the proof only for the fixed point operator.

**Lemma 7.5** (i) If \((c, \eta) \ll^\Delta k f \) and \((c', \eta) \ll^\theta d, \) then \( \text{Push} c' \triangleright c, \eta \ll^\Delta k f d. \) (ii) If \( \eta \ll^\pi \gamma \) and \( n < |\pi|, \) then \( \text{Access} n, \eta \ll^\Delta k \gamma \downarrow n. \)
(iii) If \((c, \eta) \vdash^\emptyset \theta \rightarrow f\) then \((\text{Fix} \triangleright c, \eta) \vdash_k^\emptyset_\theta Y[\theta] f\).

(iv) If \((c_i, \eta) \vdash^\text{int} \theta d_i\) for all \(i \in \{1, \ldots, n\}\), then
\[
(\text{Push } c_n \triangleright \ldots \triangleright \text{Push } c_1 \triangleright \text{Frame } \theta^n, \eta) \vdash^\text{int}_k \theta n d_1, \ldots, d_n.
\]

(v) If \((c, \eta) \vdash^\emptyset \theta \times \theta' (d_0, d_1)\), then
\[
(\text{Push } \text{Fst} \triangleright c, \eta) \vdash_k^\emptyset d_0 \text{ and } (\text{Push } \text{Snd} \triangleright c, \eta) \vdash_k^\emptyset d_1.
\]

(vi) If \((c, \eta) \vdash^\emptyset \theta (d_0, d_1)\) and \((c', \eta) \vdash^\text{int} \theta d\), then
\[
(\text{Push } c' \triangleright c, \eta) \vdash_k^\emptyset \theta (\lambda z. \text{if } z = 0 \text{ then } d_0 \text{ else } d_1) d.
\]

\textbf{Proof.} Let \(\alpha = (c, \eta)\), let \(\alpha' = (\text{Fix} \triangleright c, \eta)\), and \(d = Y[\theta] f\). Our goal is to prove that \(\alpha \vdash_k^\emptyset \theta \rightarrow f\) implies \(\alpha' \vdash_k^\emptyset d\), we proceed by induction over \(k\). The case \(k = 0\) is trivial since \(\emptyset_0 = \Gamma \times \llbracket \emptyset \rrbracket\). Now we assume \(\alpha \vdash_k^\emptyset \theta \rightarrow f\) and prove \(\alpha' \vdash_{k+1}^\emptyset d\). We take \((l, s) \in \Gamma^\emptyset (d)^l\text{-}S\) with \(l \leq k + 1\), and prove \((\alpha', s) \in R_l\).

We have two cases depending on whether \(l \leq k\) or \(l = k + 1\). In the first case we use our inductive hypothesis \(\alpha' \vdash_k^\emptyset d\) to obtain \((\alpha', s) \in R_l\). Now assume \(l = k + 1\). We have \((k + 1, s) \in \Gamma^\emptyset (d)^{k+1}\text{-}S\) and hence \((k, s) \in \Gamma^\emptyset (f d)^{k+1}\text{-}S\). Since \(d = f d\) we obtain \((k, s) \in \Gamma^\emptyset (f d)^{k+1}\). By inductive hypothesis we have \(\alpha' \vdash_{k+1}^\emptyset d\), and hence \((k, \alpha' :: s) \in \Gamma^\emptyset (\theta (f) \rightarrow \theta) (f)^{k+1}\text{-}S\). We had \(\alpha \vdash_k^\emptyset \theta \rightarrow f\) by assumption, and hence \((\alpha, \alpha' :: s) \in R_{k+1}\). Since \((\alpha', s) \triangleright (\alpha, \alpha' :: s)\) we obtain \((\alpha', s) \in R_{k+1} = R_l\).

The fundamental lemma of the logical relation, which states that the compilation of a typing derivation is an approximation of its semantics, is a direct consequence of Lemma 7.5.

\textbf{Lemma 7.6 (Operational approximation of compiled code)} For all \(i \in \mathbb{N}\), if \(\eta \vdash^\text{int} \gamma\), then \(\llbracket t \rrbracket_{\pi, \theta, \eta} \vdash^\text{int} \llbracket \pi \vdash t : \theta \rrbracket \gamma\).

Here the relation \(\vdash^\text{int}\) is defined as a pointwise extension similarly to 5.3. Lemmas 7.6 and 7.2 lead to the following result.

\textbf{Lemma 7.7} If \(t\) is a closed term, and \(\llbracket \cdot \vdash t : \text{int} \rrbracket () = \bot\), then the configuration \(\llbracket \cdot \vdash \text{int} \rrbracket ((\cdot t)_0, \text{int} [], s)\) diverges for any stack \(s\).

Finally, as a consequence of lemmas 5.8 and 7.7, we can state a compiler correctness theorem.

\textbf{Theorem 7.8 (Compiler correctness)} Suppose \(t\) is a closed term of type \text{int}. If \(\llbracket \cdot \vdash t : \text{int} \rrbracket () = \nu \eta m\), then \(\llbracket \cdot t \rrbracket_{\text{int} [], m}, [] \mapsto^\star (\cdot (\text{Const } m, \eta), []), \) for some \(\eta \in H\). Otherwise, if \(\llbracket \cdot \vdash t : \text{int} \rrbracket () = \bot\), then \(\llbracket \cdot t \rrbracket_{\text{int} [], m}, [] \) diverges.

\section{Formalization}

All the results presented in this paper has been completely formalized in the proof assistant Coq (version 8.4pl5 with Ssreflect 1.5). The formalization is constructive, as we do not assume any classical axiom. We invite the curious reader to download [25] and explore the formalization as it complements the content of this article.

Our formal development is based on a domain-theory library by Benton et al. [6] that provided us with the basis to formalize the denotational semantics of the language; our formalization would have taken much more time without that library. As useful as it was, we found some shortcomings that we turned into extensions:
• The original “extension” function, named \texttt{kleisli}, of the lifting monad has type
\[ \texttt{kleisli} : (P \to Q_{\perp}) \to (P_{\perp} \to Q_{\perp}). \]
This operator is adequate for a call-by-value language; in our setting, however this is not enough, the semantics of the conditional asks for the following operator \texttt{gkleisli}:
\[ (P \to D) \to (P_{\perp} \to D) \]
where \( D \) is any pointed cpo (not necessarily obtained through lifting).

• A formalization of \( n \)-ary morphisms and finite products used to implement the semantics of \( n \)-ary operators and to prove some results about them.

• A variety of results regarding Cartesian closed categories, cpos and the computation of least upper bounds.

The formalization is modular; we defined Galois connections, biorthogonality and step-indexed families in full generality and then we instantiated this modules appropriately. We also extended a formalization of sequences (finite and infinite) originally written in [18]. Our own development (excluding the domain-library) has 6134 lines of code in total, 2096 of which are specifications and 4038 are proofs.

9 Conclusion and Further Work

We have proved the correctness of a compiler for a call-by-name functional language by means of logical relations defined using biorthogonality and step-indexing. This abstract setting provides a certain degree of flexibility with respect to modifications of the execution environment and it is also modular with respect to the constructors of the language; in particular, the use of step-indexing enabled us to deal with inductive proofs in the presence of recursion.

This approach is similar to [4] but with some important differences due to the order of evaluation (call-by-name instead of call-by-value) and the nature of the abstract machine (KAM rather than the SECD machine). The lack of difference between values and terms in the call-by-name setting turns our logical relations simpler and more intuitive than those in [4]: there is only one kind of approximations on the operational side (closures) and there is no need for a “monadic lifting”. In addition, our definition of the orthogonal operators is simpler since there is no need to parameterize them using environments or any other kind of value.

As future work we plan to extend the source language by enriching the type system and adding new constructors. For example, in [26] we proved the correctness of a compiler for a higher-order imperative language with respect to the big-step operational semantics of the source language; it would be interesting to obtain a relational proof of compiler correctness. We also intend to investigate the application of this technique to lazy functional languages targeting the Sestoft abstract machine [28] or the STG machine [13]. We are also interested in applying the method to other models of execution closer to real assembly code.

References


Canonical HybridLF: Extending Hybrid with Dependent Types

Roy L. Crole1 Amy Furniss2

Dept of Computer Science, University of Leicester, University Road, Leicester, LE1 7RH, U.K.

Abstract

We introduce Canonical HybridLF (CHLF), a metalogic for proving properties of deductive systems, implemented in Isabelle HOL. CHLF is closely related to two other metalogics. The first is the Edinburgh Logical Framework (LF) by Harper, Honsell and Plotkin. The second is the Isabelle HOL Hybrid system developed by Ambler, Crole and Momigliano which supports use of Higher-Order Abstract Syntax (HOAS) through an un-typed lambda calculus.

Historically there are two problems with HOAS: its incompatibility with inductive types and the presence of exotic terms. Hybrid provides a partial solution to these problems whereby HOAS functions that include metalogic bound variables are automatically converted to a machine-friendly de Bruijn representation hidden from the user.

The key innovation of CHLF is the replacement of the un-typed lambda calculus with a dependently-typed lambda calculus in the style of LF. CHLF allows signatures containing constants representing the judgements and syntax of an object logic, together with proofs of metatheorems about its judgements, to be entered using a HOAS interface. Proofs that metatheorems defined in the signature are valid are created using the M2 metalogic of Schurmann and Pfenning.

We make a number of advances over existing versions of Hybrid: we now have the utility of dependent types; the unitary bound variable capability of Hybrid is now potentially finitary; and the old method of indicating errors using special elements of core datatypes is replaced with a more streamlined one that uses the Isabelle option type.

CHLF has an advantage over Twelf in that a proof that a metatheorem holds is explicitly stated, rather than being automatically generated and hidden from the user. This brings difficulties, however, as the search for a proof can be arduous when executed by hand. It is future work to develop domain specific automatic search procedures.

Keywords: dependent types, HOAS, logical frameworks, metalogical reasoning, variable binding

1 Introduction

The first author, Crole, along with Ambler and Momigliano, developed the HYBRID system [1]. This is an implementation of Higher Order Abstract Syntax within an Isabelle HOL package. The key novel feature is that a user may write down syntax using “user friendly” named binding variables, but the package converts such syntax to “machine friendly” de Bruijn notation for its “internal reasoning”. The HYBRID system is underpinned by the untyped \( \lambda \)-calculus, and certain key conversion functions make crucial use of higher order pattern matching. Our contribution

Selected, revised papers will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Email: rlc30le.ac.uk
2 Email: mjf290le.ac.uk
is to show that the conversion technique extends not just to a simply typed setting, but in fact to a new system, Canonical HybridLF, that is dependently typed. Further, we show that the new system provides for Twelf-style reasoning with the judgements-as-types methodology.

Just as untyped λ-calculus underpins HYBRID, Canonical LF underpins Canonical HybridLF (see Sections 4 and 5 for further details). Originally we were motivated by a simple curiosity to explore the possibilities afforded by a Hybrid-style system based on dependent types (in an initial development stage, a new version of HYBRID was produced based on a simple type system). Taking this a step further we asked, given a Hybrid-style system based on an “LF style” dependent type theory, is it possible to perform Twelf-style reasoning about metatheorems in which proofs of totality are explicitly set out, rather than automatically generated and hidden from the user as in Twelf? In fact Canonical HybridLF serves as a proof-of-concept that proofs-as-types reasoning within a general purpose tactical theorem prover is possible. It is based (unsurprisingly) on Canonical LF rather than LF, and we discuss reasons for this in Section 7.

In section 2 we recall some details about Higher Order Abstract Syntax, including some well known issues. In section 3 we explain the HYBRID system which solved some of these issues. We point the reader to the original papers, and provide some necessary background reading. In section 4 we briefly recall the LF and Canonical LF logical frameworks. In section 5 we describe the main contribution of this paper, namely an implementation of Canonical LF that demonstrates that the conversion techniques at the heart of HYBRID can be extended in a smooth way from an untyped λ-calculus setting all the way to a dependently typed λ-calculus, thus meeting our main goal. In section 6 we give some examples and discuss an implementation of a logic for proofs in our system.

There is more about the theory of HYBRID in [5]. More recently a version of HYBRID was created by Martin [12] and Amy Felty. Versions of HYBRID developed in Coq appear in [3] and [9] with work by Capretta, Felty and Habli.

2 Background and Motivation

This paper concerns reasoning about deductive systems such as logics, programming languages and so on. In general we refer to a typical deductive system as an object logic. One may reason about an object logic by translating it into a metalogic and then performing reasoning in the metalogic, provided that properties of the object logic are suitably reflected in the metalogic.

In particular this paper concerns Higher-Order Abstract syntax (HOAS). This is a well-known metalogical technique that can be applied to object logics that have variable binding: variable binders of the HOAS metalogic are used to implement the variable binders of an object logic. While there has been success in using HOAS there are some issues.

One issue is that HOAS is typically not compatible with inductive definitions. Given, say, ∀v.φ in an object logic, with recursive translation $∀v.φ \rightarrow \text{forAll}(λv.\overline{φ})$ in the metalogic, the constructor forAll must have type forAll : (exp ⇒ exp) ⇒ exp. However this is not compatible with the injectivity constraints on constructors of
inductive types: for (classical) cardinality reasons forAll cannot be injective. This is well documented in the literature; see for example [1]. Another issue is the presence of exotic terms: terms that can appear in the metalogic that have no corresponding term in the object logic. Roughly speaking there are definable functions in exp ⇒ exp that are not in the image of the translation function, and this complicates the metalogical reasoning.

A number of approaches to circumventing these limitations appear in the literature. One such approach is that of Despeyroux et al [6], in which a type var of variables is introduced and used as the source type of the binder functions, thus allowing injections of (var ⇒ exp) into exp. The key disadvantage with this methodology is that substitution in the object logic is no longer implemented as β-reduction in the metalogic: an inductive definition of substitution must be used instead. Momigliano et al [13] refer to techniques such as this, in which object logic bound variables are encoded as metalogic bound variables and object logic contexts are encoded as metalogic contexts but substitution is not implemented as metalogic β-reduction, as weak HOAS. Chlipala [4] describes parametric higher-order abstract syntax, another (but related) approach to weak HOAS, in which a type parameter is introduced as the type of variables. Recently, some challenges for HOAS have been documented in [8] by Felty, Momigliano, and Pientka.

3 The Hybrid Metalogic

In HYBRID terms of any object logic are entered by the user as HOAS metalogic terms (e.g. forAll (λv.φ)) with named bound variables (i.e. v). Such terms are then automatically converted to a nameless de Bruijn form in which instances of bound variables are given by a numerical index or level. The idea is that the user can work with named terms while the machine works with equivalent nameless terms that are automatically generated. The system is a “hybrid” of named and nameless variable binding. (HYBRID utilises locally nameless de Bruijn terms [2]: Bound variables are denoted by instances of BND, and have a natural number index; free variables are indicated by instances of VAR, and are also indexed by the natural numbers.)

The original version of HYBRID provides HOAS in the form of the untyped lambda calculus, and is implemented as a package for the Isabelle theorem prover. The implementation is based around a core inductive datatype expr that implements locally nameless de Bruijn terms:

Definition 3.1 [Core HYBRID Datatype]

\[ \text{Expr} ::= \text{BND} \, \text{nat} | \text{VAR} \, \text{nat} | \text{CON}'a' | \text{ABS expr} | \text{APP expr expr} | \text{ERR} \]

The CON constructor is used to denote an instance of an object logic constant. The Expr type has a type parameter ‘a, and the elements of this type are used to specify the constants of an object logic. The constructor forAll of page 232 would be rendered in HYBRID as CON cForAll (where, for example, ‘a = cForAll | cExists specifies object logic quantifiers). The APP constructor denotes application (often written as infix $$) and the ABS constructor denotes (de Bruijn) function
abstraction. Finally the ERR constructor is a special element used to indicate if an error occurs during conversion from HOAS function to de Bruijn indices.

In HYBRID, a general HOAS term $\lambda v.e$ is written by the user as LAM $v.e$; this is legitimate Isabelle syntax. Thus forAll ($\lambda v.\overline{e}$) of page 232 would be written by the HYBRID user as (CON cforAll) $\equiv$ (LAM $v.\overline{e}$). The point is that HYBRID provides a very natural user “interface” for HOAS with named bound variables.

To explain further, LAM $v.e$ is an abbreviation for lambda($\lambda v.e$) where lambda is an Isabelle function that automatically converts $\lambda v.e$ to a de Bruijn term. lambda lies at the heart of the HYBRID system. Full details can be found in [1,5]. Here we cannot explain in detail the intuition behind the implementation of lambda, but give a brief overview and the Isabelle definitions. lambda($\lambda v.e$) is defined to be ABS (lbind 0 ($\lambda v.e$)). The main things readers need to know are

(i) $\lambda v.e$ must be a (unary) HYBRID abstraction (c.f. section 5.2).

(ii) The lbind (c.f. section 5.3) function converts $\lambda v.e$ to de Bruijn form.

To explain the notion of abstraction we first recall the adjectives dangling and level (which are well-known properties of de Bruijn terms). A variable BND $j$ in an term $e$ is said to be dangling if $j$ or less ABS nodes occur strictly between $j$ and the root of $e$. For instance, in the term $T = \text{ABS} (\text{BND} 0 \equiv \text{BND} 1)$ the bound variable instance BND 0 is not dangling because there are zero such ABS nodes, but BND 1 is indeed dangling. Terms in HYBRID have a level. A term $e$ is at level $l \geq 1$ if enclosing $e$ in $l$ ABS nodes ensures that the resulting expression has no dangling variables. For instance, the term $T$ would be at level 1 because it requires one extra enclosing ABS to ensure that no variables are dangling. Terms at level 0—with no dangling indices— are called proper terms. Now we define abstraction. A unary abstraction is, informally, an expression at level 1 in which any dangling index is replaced by a metavariable (say $v$) and then abstracted (enclosed by $\lambda v$). An example is $A = \lambda v. \text{ABS} (\text{BND} 0 \equiv v)$. This is a key idea. A direct correspondence between de Bruijn indices, such as BND 1 in $T$, and the binding mechanism of the metalogic, such as $\lambda v. \ldots v$ in $A$, is set up. In the original HYBRID [1] there is a predicate abstr that defines when its argument is a valid unary abstraction. It is implemented using an inductive relation abst. The definition is omitted here, but background details appear in [1]. The formal definition of abstr is then $\text{abstr} \equiv \text{abst} \equiv 0$.

Briefly, abstr works by recursion on $e$ over the expr constructors, removing the constructor at each call, and moving each $\lambda v$ towards the leaf nodes of $e$. The index $i$ increases when recursing over each ABS node, hence counting the nodes. At a leaf, $\lambda v. v$ is deemed an abstraction as is $\lambda v. \text{VAR} n$. In the case of a leaf $\lambda v. \text{BND} j$, $i$ is now equal to the number of ABS nodes between the leaf and the root, and so BND $j$ is NOT dangling provided that $j < i$: hence $\lambda v. \text{BND} j$ is an abstraction. Overall abstr returns the conjunction of the leaf node results.

lbind is defined using an inductive relation lbind. The definition is omitted here, but background details appear in [1]. lbind is defined as lbind $i e \equiv e s$. lbind $i e s$.

3 The section references refer to descriptions of the analogues of abstractions and lbind for CANONICAL HYBRIDLF.
Crole and Furniss

bind works by recursion on $e$ over the $\text{expr}$ constructors, leaving each constructor in place, and moving each $\lambda v$ towards the leaf nodes of $e$. The index $i$ increases when recursing over each $\text{ABS}$ node, hence counting them. At a leaf, $\lambda v. v$ is replaced by $\text{BND} i$, the correct de Bruijn index. All other leaf nodes remain unaltered.

4 LF and Canonical LF

LF is the Edinburgh Logical Framework, formulated by Harper et al [10]. It is a dependently-typed lambda calculus, intended as a metalogic for reasoning about deductive systems. An LF type is created for each judgement of the deductive system, and a proof that any particular judgement holds is specified through an LF term that inhabits the type that corresponds to the judgement. In this way, proof checking is reduced to type-checking (which is decidable in LF by design). This is often called the judgements-as-types approach [10].

LF has a notion of canonical form, which in [10] are kinds, types and terms that are $\beta$-normal, $\eta$-long, and correspond to terms of the object logic. It also has notions of definitional equality, which define when two kinds, types or terms are definitionally equal. All kinds, types and terms in LF are definitionally equal to a kind, type or term in canonical form.

Watkins et al [19] give a canonical presentation of LF in which only kinds, terms and types in canonical form can be formed due to restrictions on the grammar, reducing definitional equality to syntactic equality. This ensures that only LF terms that actually represent terms of the object logic can exist, and eliminates the need to reason about definitional equality. The grammar of canonical LF is as follows:

Definition 4.1

$$
\begin{align*}
K & ::= _k \text{Type} \mid \Pi x:A.K \\
M & ::= _m R \mid \lambda x.M \\
A & ::= _a P \mid \Pi x:A.A \\
R & ::= _r x \mid c \mid R M \\
P & ::= _p a \mid P M
\end{align*}
$$

Here we use $K$ to indicate an arbitrary kind, $A$ to indicate a canonical type, $P$ to denote an atomic type, $M$ to denote a canonical term and $R$ to indicate an atomic term. Signatures $\Sigma$ consist of either the empty signature $\langle \rangle$, or a list of term constants $c$ and type constants $a$ with their types or kinds. There is no need to define definitional equality relations as Harper et al [10] do in the original LF paper, since definitional equality in canonical LF is syntactic equality. The typing rules for canonical LF are given in figure 1.

The main disadvantage of this “canonical” approach is that it requires some care when performing substitution, as we must ensure that the result of substituting a canonical term into a canonical term or type is also a canonical term or type. However Watkins et al define a notion of hereditary substitution which satisfies this requirement, and in fact we follow the definition of hereditary substitution given by Harper and Licata [11]. Here are some overview comments: We let $\iota$ range over the symbols $k$, $a$, $p$, $m$, $r$ each of which refers to one of the syntactic categories in Definition 4.1. The definition of substitution makes use of simple types defined by the grammar $\alpha ::= a \mid \alpha \rightarrow \alpha$, where $a$ ranges over type constants. We use $[M/x]_\alpha^\iota_\xi = \xi'$ to denote the hereditary substitution of the canonical term $M$
Fig. 1. Canonical LF Judgements

(category m) for free occurrences of the variable x in the expression ξ, resulting in the expression ξ'. Here, α is the simple type of M defined by type-erasure (see [11]); and both and ξ and ξ' range over the same syntactic category ι. The superscript ι plays a technical role in certain substitution rules, differentiating between substitution on atomic terms or types.

5 Canonical HybridLF

5.1 The Core Datatype

In essence, the original HYBRID is a system that provides a HOAS interface to an untyped lambda calculus. As explained in Section 3, the user’s methodology is to encode an object logic using “lambda calculus HOAS terms”. Such terms are then automatically converted to de Bruijn terms and reasoned about directly in Isabelle, using Isabelle’s higher-order logic and the Isabelle lemma construct to create proofs. The user’s methodology in CANONICAL HYBRIDLF is slightly different. CANONICAL HYBRIDLF is a system that, using an adaptation of the HYBRID approach to variable binding provides a HOAS interface to a dependently-typed lambda calculus in the style of Canonical LF. Theorems and meta-theorems are

- defined in a signature, where the HOAS interface enables the user to enter the LF signature as Isabelle functions with named bound variables (which are then converted to de Bruijn form), and

CANONICAL HYBRIDLF acts as a metalogic, permitting the user to prove properties of deductive systems. Since CANONICAL HYBRIDLF implements canonical LF it permits the use of the judgements-as-types approach to proving theorems.
Recall the five syntactic categories of Definition 4.1. Canonical HybridLF is based around five mutually-inductively defined datatypes \texttt{ctype}, \texttt{atype}, \texttt{cterm}, \texttt{aterm} and \texttt{kind}. These mutually defined datatypes yield an overall core datatype inhabited by terms corresponding to those of Definition 4.1, much as \texttt{expr} is the core datatype at the heart of Hybrid. The new core datatype is built out of \texttt{VAR} and \texttt{BND} constructors so that, ultimately, within the machine implementation variable binding is once again boiled down to de Bruijn nameless terms. However a user can still work with a HOAS-style interface with named bound variables.

**Definition 5.1 [Core Canonical HybridLF Datatype]**

\[
\text{datatype} \ (\ 'a, 'b) \text{ kind} = \text{TYPE} | \text{KPI} \ (\ 'a, 'b) \text{ ctype} "(\ 'a, 'b) \text{ kind}"
\text{and} \ (\ 'a, 'b) \text{ ctype} = \text{ATYPE} "(\ 'a, 'b) \text{ atype}"
\quad | \text{PI} "(\ 'a, 'b) \text{ ctype} "(\ 'a, 'b) \text{ ctype}"
\text{and} \ (\ 'a, 'b) \text{ atype} = \text{FCON} 'b
\quad | \text{FAPP} "(\ 'a, 'b) \text{ cterm} "(\ 'a, 'b) \text{ atype}"
\text{and} \ (\ 'a, 'b) \text{ cterm} = \text{ATERM} "(\ 'a, 'b) \text{ aterm}"
\quad | \text{ABS} "(\ 'a, 'b) \text{ ctype} "(\ 'a, 'b) \text{ cterm}"
\text{and} \ (\ 'a, 'b) \text{ aterm} = \text{VAR} \ \text{nat} | \text{BND} \ \text{nat} | \text{CON} \ 'a
\quad | \text{APP} "(\ 'a, 'b) \text{ aterm} "(\ 'a, 'b) \text{ cterm}"
\]

Notice that there are two type parameters: one parameter is used to specify the object constants and one to specify the type constants of the object logic. (Recall that the analogous core datatype \texttt{expr} in the original Hybrid–Definition 3.1 on page 233–has just a single type parameter).

**5.2 Abstractions in Canonical HybridLF**

Like Hybrid, Canonical HybridLF employs the concept of abstraction, and has function predicates that determine if certain terms are valid abstractions. There are however two main differences. The first is that we now have the concept of type abstractions as well as term abstractions. The second is that Canonical HybridLF extends the general notion of abstraction from the unary abstractions found in Hybrid to \( k \)-ary abstractions. Intuitively speaking these are Isabelle functions that bind exactly \( k \) variables, are syntactic terms and have no dangling indices.

Canonical HybridLF has four families of functions that determine if a function represents a valid term abstraction or type abstraction. By ‘families’ of functions, we mean sets of functions each with the same name apart from a numerical suffix (e.g. 12) indicating the expected arity of the input (e.g. a 12-ary abstraction). The prefix of the function indicates which syntactic category the function acts upon. The production of these functions variants numbered up to 12 is a pragmatic choice based on the trade-off between theory processing time and availability of functions for the user. Each additional function adds to the time necessary to process the Canonical HybridLF theory file in Isabelle. On the other hand, we required
variants of \texttt{ctype\_bind} (see Section 5.3) numbered up to 11 for our examples, so producing versions taking up to 12 variables seemed a reasonable step in practice.

These functions are analogous to \texttt{abstr} in the original HYBRID (see page 234). In CANONICAL HYBRIDLF these four families consist of

\begin{itemize}
\item[a] \texttt{ctype\_abstr} to \texttt{ctype\_abstr12},
\item[p] \texttt{atype\_abstr} to \texttt{atype\_abstr12},
\item[m] \texttt{cterm\_abstr} to \texttt{cterm\_abstr12}, and
\item[r] \texttt{aterm\_abstr} to \texttt{aterm\_abstr12}.
\end{itemize}

For example \texttt{cterm\_abstr} determines if an Isabelle function with one bound variable is a valid unary abstraction that corresponds to a canonical term, while \texttt{cterm\_abstr12} determines if a function with twelve bound variables is a valid abstraction that corresponds to a canonical term. The other three families of functions perform the same task for abstractions representing canonical types, atomic types and lastly atomic terms. Variants for up to twelve variables suffice in practice; there are no limits on the number of variables that can be handled so we could define \texttt{abstr} functions that allow the analysis of functions with a greater number of bound variables.

We give one example definition, of \texttt{cterm\_abstr}, and in order to do so we need an auxiliary definition, \texttt{cterm\_ordinary}:

\textbf{Definition 5.2} [auxiliary predicate: \texttt{cterm\_ordinary}]

\begin{align*}
\text{cterm\_ordinary} & \equiv \lambda e. (e = (\lambda x. \text{ATERM } x)) \\
& \lor (\exists n. e = (\lambda x. \text{ATERM } (\text{BND } n))) \\
& \lor (\exists n. e = (\lambda x. \text{ATERM } (\text{CON } n))) \\
& \lor (\exists n. e = (\lambda x. \text{ATERM } (\text{VAR } n))) \\
& \lor (\exists f g. e = (\lambda x. \text{ATERM } ((f x) \ $$g x)))) \\
& \lor (\exists f ty. e = (\lambda x. \text{ABS } (ty x) (f x)))
\end{align*}

The auxiliary predicate \texttt{cterm\_ordinary} is again a family of definitions, with twelve variants for Isabelle functions with up to twelve bound variables. It holds when a term is syntactic (i.e. not an exotic term). \texttt{cterm\_abstr} is then defined like so:

\textbf{Definition 5.3} [\texttt{cterm\_abstr}]

\begin{align*}
\text{cterm\_abstr } i (\lambda x. \text{ATERM } x) & = \text{True} \\
\text{cterm\_abstr } i (\lambda x. \text{ATERM } (\text{CON } a)) & = \text{True} \\
\text{cterm\_abstr } i (\lambda x. \text{ATERM } (\text{BND } n)) & = (n < i) \\
\text{cterm\_abstr } i (\lambda x. \text{ATERM } (\text{VAR } n)) & = \text{True} \\
\text{cterm\_abstr } i (\lambda x. \text{ATERM } ((f x) $$g x))) & = \\
& (\text{aterm\_abstr } i f \land \text{cterm\_abstr } i g) \\
\text{cterm\_abstr } i (\lambda x. \text{ABS } (t x) (f x)) & = (\text{ctype\_abstr } i t \\
& \land \text{cterm\_abstr } (i + 1) f) \\
\neg \text{cterm\_ordinary } f & \implies \text{cterm\_abstr } i f = \text{False}
\end{align*}

Note that \texttt{cterm\_ordinary} is used as a guard in the last equation, which returns \texttt{False} when the term is exotic.
5.3 Conversion to de Bruijn in Canonical HybridLF

Recall the functions lambda and lbind from section 3. These functions implement the automatic conversion of HOAS expressions with named Isabelle binders $v$ to de Bruijn form. A key contribution of the current paper is to show that the higher order pattern matching techniques used in implementing these functions in HYBRID transfer smoothly to an analogous implementation of Canonical LF, thus yielding the Canonical HybridLF system.

Conversion in Canonical HybridLF from HOAS expressions to de Bruijn terms is performed by families of functions $\text{ctype}$ bind to $\text{ctype}$ bind12, $\text{atype}$ bind to $\text{atype}$ bind12, $\text{cterm}$ bind to $\text{cterm}$ bind12, and $\text{aterm}$ bind to $\text{aterm}$ bind12.

To create these functions we define families of functions $\text{ctype}$ bind' to $\text{ctype}$ bind'12, $\text{atype}$ bind' to $\text{atype}$ bind'12, $\text{cterm}$ bind' to $\text{cterm}$ bind'12, and finally $\text{aterm}$ bind' to $\text{aterm}$ bind'12. The former unprimed functions are analogues of lambda, and the primed functions are analogues of lbind. Here is the definition of $\text{cterm}$ bind':

**Definition 5.4** [cterm\_bind']

$\text{cterm}$ bind' $i (\lambda x. \text{ATERM } x) = \text{Some } (\text{ATERM } (\text{BND } i))$

$\text{cterm}$ bind' $i (\lambda x. (\text{ATERM } (\text{BND } k) )) = \text{Some } (\text{ATERM } (\text{BND } k))$

$\text{cterm}$ bind' $i (\lambda x. (\text{ATERM } (\text{VAR } n ))) = \text{Some } (\text{ATERM } (\text{VAR } n))$

$\text{cterm}$ bind' $i (\lambda x. \text{ATERM } (\text{CON } a)) = \text{Some } (\text{ATERM } (\text{CON } a))$

$\text{cterm}$ bind' $i (\lambda x. \text{ATERM } (\text{APP } (F x) (G x))) = (\text{case }$

$(\text{aterm\_bind'} i F) \text{ of Some atm } \Rightarrow (\text{case } (\text{cterm\_bind'} i G)  

\text{ of Some ctm } \Rightarrow \text{Some } (\text{ATERM } (\text{APP atm ctm})) | \text{None } \Rightarrow \text{None})$

$\text{cterm\_bind'} i (\lambda x. \text{ABS } (ty x) (F x)) = (\text{case }$

$(\text{cty\_bind' } i ty) \text{ of Some t } \Rightarrow (\text{case } (\text{cterm\_bind'} (i + 1) F)  

\text{ of Some m } \Rightarrow \text{Some } (\text{ABS } t m) | \text{None } \Rightarrow \text{None})$

$\text{None } \Rightarrow \text{None})$

$\neg\text{cterm\_ordinary expr } \Rightarrow \text{cterm\_bind'} i expr = \text{None}$

The natural number argument $i$ of $\text{cterm\_bind'}$ tracks how many ABS nodes have been recursed over. Note that $\text{cterm\_bind'}$ returns a cterm option, returning None in the last equation when its argument is not a syntactic term, and Some $m$ when the result is a canonical term $m$. This is in contrast to the original HYBRID, which makes use of an ERR element of the core expr datatype to indicate that an error has occurred (see page 233): coding and error handling are now slicker.

The definition of $\text{cterm\_bind}$ (an analogue of lambda) is as follows:
Definition 5.5 [cterm_bind]

cterm binds $t \ e \equiv \text{if} \ \text{cterm abstr} \ 0 \ e \ \text{then} \ (\text{case} \ (\text{cterm bind}' \ 0 \ e)\ \text{of} \ \text{Some} \ \ e' \Rightarrow \text{Some} \ (\text{ABS} \ t \ e') \ | \ \text{None} \Rightarrow \text{None}) \ \text{else} \ \text{None}$

cterm bind takes as parameters a canonical type $t$ (for the type of the bound variable) and a function $e$ to be converted to de Bruijn form. This conversion is performed by calling the cterm bind' function on $e$ with the initial parameter of 0 for the number of binders recursed over. The aterm bind function is defined similarly, with an aterm bind' function performing the actual work of converting HOAS functions to de Bruijn form.

cctype bind is defined like so:

Definition 5.6 [ctype_bind]

ctype binds $t \ e \equiv \text{if} \ \text{ctype abstr} \ 0 \ e \ \text{then} \ (\text{case} \ (\text{ctype bind}' \ 0 \ e)\ \text{of} \ \text{Some} \ \ e' \Rightarrow \text{Some} \ (\text{PI} \ t \ e') \ | \ \text{None} \Rightarrow \text{None}) \ \text{else} \ \text{None}$

Again, $t$ is a canonical type and $e$ is function to be converted to de Bruijn form. In variants of cctype bind that convert a function with a number of variables greater than one, there are more than one arguments for the type of binders, more than one enclosing binder is added to the converted term, and the argument types of the types of binders are functions with increasing numbers of bound variables. For instance, cctype bind3 is defined as follows:

Definition 5.7 [ctype_bind3]

ctype binds $t1 \ t2 \ t3 \ e \equiv \text{if} \ \text{ctype abstr3} \ 0 \ e \wedge \text{ctype abstr} \ 0 \ t2 \\
\wedge \text{ctype abstr2} \ 0 \ t3 \ \text{then} \ (\text{case} \ (\text{ctype bind}' \ 0 \ t2)\ \text{of} \ \text{Some} \ t2' \Rightarrow \\
(\text{case} \ (\text{ctype bind}'2 \ 0 \ t3) \ \text{of} \ \text{Some} \ t3' \Rightarrow (\text{case} \ (\text{ctype bind}'3 \ 0 \ e) \\
\text{of} \ \text{Some} \ e' \Rightarrow \text{Some} \ (\text{PI} \ t1 \ (\text{PI} \ t2' \ (\text{PI} \ t3' \ e')) \ | \ \text{None} \Rightarrow \text{None}) \ | \ \text{None} \Rightarrow \text{None}) \ \text{else} \ \text{None}$

Note that three PI binders are added to the start of the converted term $e'$, that the type of $t1$ is $('a, 'b) \ cctype$, the type of $t2$ is $('a, 'b) \ aterm \rightarrow ('a,'b) \ cctype$ and the type of $t3$ is $('a, 'b) \ aterm \rightarrow ('a, 'b) \ cctype$. The types $t2$ and $t3$ of the latter two binders are given as functions because the variables bound in the preceding binders may appear within them.

5.4 Typing, kinding and substitution in Canonical HybridLF

Typing, kinding and substitution in Canonical HybridLF are carried out by a number of mutually-defined functions. Kinds are determined to be valid by the validkind function, while types are determined to be valid by the validtype function. Kinds are assigned to atomic types by the atom kokindof function, while types are assigned to atomic and canonical types by the atom typeof and canon typeof functions respectively. These functions correspond to the typing and kinding rules of canonical LF.
Substitution is performed by a number of functions, mostly carrying out substitution for terms or types from a syntactic category of the grammar. The first of these functions is \texttt{ctype_subst_t}, which substitutes a canonical term for free variables in a canonical type, corresponding to \([M/x]_{\alpha}A\). The \texttt{atype_subst_t} function performs substitution of a canonical term for free instances of a variable in atomic type, corresponding to \([M/x]_{\alpha}P\). The \texttt{cterm_subst_t} function substitutes a canonical term for a free variable in a canonical term, corresponding to \([M/x]_{\alpha}M\). Substitution for atomic terms is carried out by two separate functions: \texttt{aterm_subst_t} and \texttt{aterm_can_subst_t}. \texttt{aterm_subst_t} performs substitution of a canonical term for free variables in an atomic term, resulting in an atomic term. It corresponds to \([M/x]_{\alpha}P\). \texttt{aterm_can_subst_t} substitutes a canonical term for free instances of a variable in an atomic term, resulting in a canonical term, corresponding to \([M/x]_{\alpha}R = M' : \alpha\).

To explain these functions we first make some general comments. The first five arguments of the typing and substitution functions are the same; see figure 2. Since Isabelle requires all functions to terminate, we introduce a numerical recursion-depth argument to ensure that this is so. Note that all of the functions have a case for when this argument is zero which simply returns \texttt{None} to indicate failure. The cases for when this parameter is non-zero all pattern-match on \texttt{Suc q} for some \texttt{q}, distinguishing them from the zero case, and recursive calls within the body of the functions all give \texttt{q} as the first parameter, ensuring that this decreases with each recursive call. The second parameter is a context, while the third and fourth are the signature, split into object constants and type constants. The fifth parameter is the binding environment, an Isabelle list of the canonical types of the enclosing binders that allows us to type bound variables.

In \texttt{canon_typeof} and \texttt{atom_typeof}, the sixth parameter is the canonical term or atomic term to determine the type of, while the sixth parameter in the substitution functions is the canonical term that we are substituting for free variables. The seventh parameter in the substitution functions is a natural number, the number of the variable to substitute for. The eighth is the term into which we are substituting.

We complete this section by giving some examples of the code for these functions. The \texttt{atom_kindof} function computes the kind of an atomic type:
Definition 5.8 [atom kindof]

atom kindof 0 ctx sig_t sig_k bnd a = None
atom kindof (q + 1) ctx sig_t sig_k bnd (FCON a) = (case sig_k lookup
  sig_k a of Some k ⇒ (if kind_level 0 k then Some k else None)
  | None ⇒ None)
atom kindof (q + 1) ctx sig_t sig_k bnd (FAPP p m) = (case atom kindof
  q ctx sig_t sig_k bnd p of Some (KPI a k) ⇒ (case canon typeof q ctx sig_t
  sig_k bnd m of Some a ⇒ kind subst bv q ctx sig_t sig_k bnd m 0 0 k
  | None ⇒ None) | None ⇒ None)

Notice that the single function atom kindof provides an implementation of two
rules that are found in figure 1. The second definitional function equation (that
is, the one for FCON a) corresponds to the LF kinding rule CON_AT_KIND and the
third definitional function equation (for FAPP p m) corresponds to the LF rule
APP_AT_KIND. Remember that the first argument is used to ensure the termination
of the atom kindof function, with the base case of 0 indicating failure to determine
a kind. In the case of a constant FCON a, a look-up is made to see if a is declared in
the signature, and if so a check is made that the kind is of level 0, a “proper kind”.
In the case of an application FAPP p m, for example where atom kindof succeeds,
the kind of p and the type of m are extracted, then the hereditary substitution of
m into the kind of p completes the computation of the kind.

The canon typeof function computes the type of a canonical term:

Definition 5.9 [canon typeof]

canon typeof 0 ctx sig_t sig_k bnd m = None
canon typeof (q + 1) ctx sig_t sig_k bnd (ATERM r) =
  atom typeof q ctx sig_t sig_k bnd r
canon typeof (q + 1) ctx sig_t sig_k bnd (ABS a' m) =
  (case canon typeof q ctx sig_t sig_k (a' # bnd) m of Some a ⇒
    (if ctype_level 0 a' then Some (PI a' a) else None) | None ⇒ None)

Like atom kindof, the base case in canon typeof returns None when the recursion-
depth limiting first parameter is 0. The second equation simply calls the atom typeof
function when the last parameter is an atomic type wrapped in the ATERM construc-
tor. The third equation (for ABS a' m) corresponds to LF typing rule ABS_CAN TY, and
determines the type of the body m of the abstraction (updating the binding en-
vironment with the type of the binder a’) and returning either None (if no type could
be computed for m or a’ is not a proper type) or Some π-type with the computed
type of m as its body.

The atom typeof function computes the type of an atomic term:
Definition 5.10 [atom_typeof]

\[
\begin{align*}
\text{atom_typeof} & \ 0 \ ctx \ sig_t \ sig_k \ bnd \ r = \text{None} \\
\text{atom_typeof} & \ (q + 1) \ ctx \ sig_t \ sig_k \ bnd \ (\text{VAR} \ v) = \\
& \quad \quad \quad \text{(case ctx\_lookup ctx v of Some t \Rightarrow (if ctyp_level 0 \ t \ then \ Some \ t \ else \ None) | None \Rightarrow None)} \\
\text{atom_typeof} & \ (q + 1) \ ctx \ sig_t \ sig_k \ bnd \ (\text{BND} \ b) = \\
& \quad \quad \quad \text{(case bndenv\_lookup bnd b of Some t \Rightarrow (if ctyp_level 0 \ t \ then \ Some \ t \ else \ None) | None \Rightarrow None)} \\
\text{atom_typeof} & \ (q + 1) \ ctx \ sig_t \ sig_k \ bnd \ (\text{CON} \ c) = \\
& \quad \quad \quad \text{(case sig_t\_lookup sig_t c of Some t \Rightarrow (if ctyp_level 0 \ t \ then \ Some \ t \ else \ None) | None \Rightarrow None)} \\
\text{atom_typeof} & \ (q + 1) \ ctx \ sig_t \ sig_k \ bnd \ (\text{APP} \ r \ m) = \\
& \quad \quad \quad \text{(case atom\_typeof q ctx sig_t sig_k bnd of Some (PI a' a) \Rightarrow} \\
& \quad \quad \quad \quad \text{(case canon\_typeof q ctx sig_t sig_k bnd m of Some a' \Rightarrow} \\
& \quad \quad \quad \quad \quad \text{ctype\_subst\_bv q ctx sig_t sig_k bnd m 0 0 a | None \Rightarrow None)} | \text{None \Rightarrow None)}
\end{align*}
\]

The second equation (for \text{VAR} \ v) corresponds to typing rule \text{VAR\_AT\_TY}, the fourth equation (for \text{CON} \ c) corresponds to \text{CON\_AT\_TY} and the fifth equation (for \text{APP} \ r \ m) corresponds to \text{APP\_AT\_TY}.

As an example of the substitution functions we show \text{aterm\_can\_subst\_fv}. This is the function that handles the previously mentioned case in which a canonical term is substituted into an atomic term (which may possibly be the first operand of an application), potentially resulting in a \(\beta\)-redex.

Definition 5.11 [aterm\_can\_subst\_fv]

\[
\begin{align*}
\text{aterm\_can\_subst\_fv} & \ 0 \ ctx \ sig_t \ sig_k \ bnd \ m \ v \ atm = \text{None} \\
\text{aterm\_can\_subst\_fv} & \ (l + 1) \ ctx \ sig_t \ sig_k \ bnd \ m \ v \ (\text{VAR} \ v') = \\
& \quad \quad \quad \text{(if v = v' then Some m else Some (ATERM (VAR v')))} \\
\text{aterm\_can\_subst\_fv} & \ (l + 1) \ ctx \ sig_t \ sig_k \ bnd \ m \ v \ (\text{BND} \ b) = \text{Some (ATERM (BND b))} \\
\text{aterm\_can\_subst\_fv} & \ (l + 1) \ ctx \ sig_t \ sig_k \ bnd \ m \ v \ (\text{CON} \ c) = \text{Some (ATERM (CON c))} \\
\text{aterm\_can\_subst\_fv} & \ (l + 1) \ ctx \ sig_t \ sig_k \ bnd \ m \ n \ (\text{APP} \ r \ m2) = \\
& \quad \quad \quad \text{(case aterm\_can\_subst\_fv l ctx sig_t sig_k bnd m n r of Some (ABS t m1') \Rightarrow} \\
& \quad \quad \quad \quad \text{(case cterm\_subst\_fv l ctx sig_t sig_k bnd m n m2 of Some m2' \Rightarrow} \\
& \quad \quad \quad \quad \quad \text{cterms\_subst\_bv l ctx sig_t sig_k bnd m2' 0 0 m1' \mid \text{None \Rightarrow None}) | \text{None \Rightarrow None})}
\end{align*}
\]

The other substitution functions are defined similarly.

6 Example Signatures and Proof System

As an initial example we looked at the raw \text{CANONICAL HYBRIDLF} signature (omitted) for the simply-typed lambda calculus, based on an example from the Twelf documentation [17].

Constructors of a \text{t\_cons} datatype specify the type constants and constructors of a \text{o\_cons} datatype the object constants of the simply-typed lambda calculus.
The signature is split into two parts. One part, \texttt{sig.kind}, contains type constants and their corresponding kinds. The other part, \texttt{sig.type.option}, contains object constants and their corresponding types. The type of \texttt{sig.type.option} is

\[
(o\text{-}cons \times (o\text{-}cons, t\text{-}cons)) \text{ctype option list}
\]

a list of pairs of object constant symbols and \texttt{ctype options}, meaning that the \texttt{option} element of the type must be removed by the user before the signature can be used (fortunately it is simple to write a function that does this). The metatheorem \texttt{pres} defined in this example is that of \textit{type preservation during evaluation}. A proof that the type preservation metatheorem represented by the \texttt{pres} type holds would be derived in the \texttt{M2} metalogic which we briefly describe in the next section.

6.1 The Logic \texttt{M2} in Canonical HybridLF

\texttt{Canonical HybridLF} employs the \texttt{M2} metalogic of Schürmann and Pfenning [15] to prove that theorems in the signature are valid. The principal reason that we make use of the \texttt{M2} metalogic is its relative simplicity, and the fact that it is powerful enough to prove a range of metatheorems. The more complicated \texttt{M2+} metalogic allows reasoning about open terms, which would be a desirable enhancement to \texttt{Canonical HybridLF}, but we leave the implementation of this to future work.

\texttt{M2} is a first-order sequent calculus with proof terms, where the proof terms of a complete derivation form a total function from universally-quantified variables to existentially-quantified variables. Formulae in \texttt{M2} have the form

\[
\forall x_1:A_1 \ldots \forall x_k:A_k, \exists x_{k+1}:A_{k+1} \ldots \exists x_m:A_m, \top
\]

where \(A_1\) to \(A_m\) are valid LF types, \(x_1 \ldots x_k\) and \(x_{k+1} \ldots x_m\) are valid contexts, quantifiers range over closed LF objects from the signature that the formula is defined for and the \(\top\) symbol stands for truth. Such formulae correspond to \textit{totality} assertions in the \textit{schema-checking} approach employed by Twelf in which variables of the meta-theorem are designated as inputs and outputs, and the system checks that for every input there exists an output. Wang and Nadathur [18] formalise the connection between Twelf schema-checking operations and \texttt{M2} derivations. \texttt{M2} is limited in that it can only reason about closed objects, not open objects (i.e. objects with free variables); Schürmann [16] introduces a further logic \texttt{M2+} that allows reasoning about open terms in a non-empty context.

\texttt{M2} has two judgements: \(\rightarrow\) and \(\rightarrow_{\Sigma}\). The \(\rightarrow\) judgement determines derivability in \texttt{M2}, while \(\rightarrow_{\Sigma}\) is an additional judgement that performs case analysis on the LF signature, attempting to unify a variable in the context with terms from the signature. If the terms unify to produce an MGU \(\sigma\) then \(\rightarrow_{\Sigma}\) requires that a derivation of \(\rightarrow\) exists for the goal formula after \(\sigma\) has been applied.

To define the proof terms of \texttt{M2}, we must first define type synonyms for substitutions, contexts, formulae and patterns. A substitution consists of pairs relating a variable number to be substituted for to the term that is to be substituted for it. Contexts consist of pairs of a free variable number and the type of the free variable. Formulae are given by a pair of contexts - the first containing universally quantified
input variables, and the second containing existentially quantified output variables. Patterns are simply terms.

We then define the datatype representing $M_2$ proof terms (definition omitted). In a complete $M_2$ derivation these proof terms form a total function from universally-quantified variables in the formula to existentially-quantified variables.

We implement the proof rules of $M_2$ as a pair of Isabelle inductive relations: derivation (corresponding to $\rightarrow$) and sig derivation (corresponding to $\rightarrow_{\Sigma}$). Space prevents us from a detailed exposition of this important aspect of our work.

7 Conclusions and future work

Our primary goal, asking if the techniques of Hybrid can be migrated to a dependently typed setting, has been answered affirmatively.

In Canonical HybridLF we make a number of advances over the existing Hybrid systems. All of these perform as they should, and they enable the use of an additional way of reasoning about deductive systems. The primary gain is the introduction of typing in the form of a dependently-typed lambda calculus, whereas previously we had only an un-typed lambda-calculus. Another key advantage with Canonical HybridLF over earlier incarnations of Hybrid is the ability to convert HOAS functions with more than one bound variable (the finitary abstractions). The method by which we achieve this, by creating families of binding functions, may initially seem like a ‘brute force’ approach. However, it is important to remember that Isabelle does not support variadic functions. We therefore do not lose anything by defining a binding function for each arity of HOAS function. That said, it will be interesting in the future to find more radical ways in which this issue can be circumvented, or even implemented in a system other than Isabelle HOL (recall that there are Coq implementations of Hybrid [3].)

In the original Hybrid HOAS terms are converted to de Bruijn terms and reasoned about directly in Isabelle, using Isabelle’s higher-order logic and the Isabelle lemma construct to create proofs. In Canonical HybridLF the theorems and meta-theorems are defined in the signature, and proved correct using the proof rules of $M_2$. The main drawback to this approach is that the search for a proof of correctness is long and tedious, whereas in Twelf this proof search is carried out automatically. As it stands, Canonical HybridLF is not so suitable for practical theorem proving, as it lacks automation of unification and proofs of totality (which can be very long and unwieldy to create by hand).

However, the requirement to explicitly set out a proof is in some ways also an advantage, as there is no need to trust an automatic procedure to perform the proof for the user. It would be possible to obtain the best aspects of both approaches by constructing a system that automatically searches for a proof, but that states the proof when it is found. The results of proof search could then be ‘certified’ in Canonical HybridLF to check their validity.

The key factors behind the choice of Canonical LF rather than LF as the basis for our system are the simplicity of equality in Canonical LF and some technical issues regarding termination of the unification algorithm that we employ for LF. This will be discussed in future work.
References


