

EPTCS 301

Proceedings of the
**Sixth Workshop on
Proof eXchange for Theorem Proving**

Natal, Brazil, August 26, 2019

Edited by: Giselle Reis and Haniel Barbosa

Published: 23rd August 2019
DOI: 10.4204/EPTCS.301
ISSN: 2075-2180
Open Publishing Association

Preface

This volume of EPTCS contains the proceedings of the Sixth Workshop on Proof Exchange for Theorem Proving (PxTP 2019), held on 26 August 2019 as part of the CADE-27 conference in Natal, Brazil.

The PxTP workshop series brings together researchers working on various aspects of communication, integration, and cooperation between reasoning systems and formalisms, with a special focus on proofs.

The progress in computer-aided reasoning, both automated and interactive, during the past decades, made it possible to build deduction tools that are increasingly more applicable to a wider range of problems and are able to tackle larger problems progressively faster. In recent years, cooperation between such tools in larger systems has demonstrated the potential to reduce the amount of manual intervention.

Cooperation between reasoning systems relies on availability of theoretical formalisms and practical tools to exchange problems, proofs, and models. The PxTP workshop series strives to encourage such cooperation by inviting contributions on all aspects of cooperation between reasoning tools, whether automatic or interactive, including the following topics:

- applications that integrate reasoning tools (ideally with certification of the result);
- interoperability of reasoning systems;
- translations between logics, proof systems, models;
- distribution of proof obligations among heterogeneous reasoning tools;
- algorithms and tools for checking and importing (replaying, reconstructing) proofs;
- proposed formats for expressing problems and solutions for different classes of logic solvers (SAT, SMT, QBF, first-order logic, higher-order logic, typed logic, rewriting, etc.);
- meta-languages, logical frameworks, communication methods, standards, protocols, and APIs related to problems, proofs, and models;
- comparison, refactoring, transformation, migration, compression and optimization of proofs;
- data structures and algorithms for improved proof production in solvers (e.g. efficient proof representations);
- (universal) libraries, corpora and benchmarks of proofs and theories;
- alignment of diverse logics, concepts and theories across systems and libraries;
- engineering aspects of proofs (e.g. granularity, flexiformality, persistence over time);
- proof certificates;
- proof checking;
- mining of (mathematical) information from proofs (e.g. quantifier instantiations, unsat cores, interpolants, ...);
- reverse engineering and understanding of formal proofs;
- universality of proofs (i.e. interoperability of proofs between different proof calculi);
- origins and kinds of proofs (e.g. (in)formal, automatically generated, interactive, ...);
- Hilbert's 24th Problem (i.e. what makes a proof better than another?);
- social aspects (e.g. community-wide initiatives related to proofs, cooperation between communities, the future of (formal) proofs);

- applications relying on importing proofs from automatic theorem provers, such as certified static analysis, proof-carrying code, or certified compilation;
- application-oriented proof theory;
- practical experiences, case studies, feasibility studies;

Previous editions of the workshop took place in Wroclaw (2011), Manchester (2012), Lake Placid (2013), Berlin (2015) and Brasília (2017).

This edition of the workshop received submissions of three regular papers, two extended abstracts and two presentation-only extended abstracts. All submissions were evaluated by at least three anonymous reviewers and one paper went through two rounds of reviewing. Five papers were accepted in the post-proceedings. The presentation-only extended abstracts were:

- DRAT-based Bit-Vector Proofs in CVC4

Authors: Alex Ozdemir, Aina Niemetz, Mathias Preiner, Yoni Zohar and Clark Barrett.

Published at: *SAT 2019*.

Abstract: Many state-of-the-art Satisfiability Modulo Theories (SMT) solvers for the theory of fixed-size bit-vectors employ an approach called bit-blasting, where a given formula is translated into a Boolean satisfiability (SAT) problem and delegated to a SAT solver. Consequently, producing bit-vector proofs in an SMT solver requires incorporating SAT proofs into its proof infrastructure. In this paper, we describe three approaches for integrating DRAT proofs generated by an off-the-shelf SAT solver into the proof infrastructure of the SMT solver CVC4 and explore their strengths and weaknesses. We implemented all three approaches using CryptoMiniSat as the SAT back-end for its bit-blasting engine and evaluated performance in terms of proof-production and proof-checking.

- Modularity Meets Forgetting: A Case Study with the SNOMED CT Ontology

Authors: Jieying Chen, Ghadah Abdulrahman S Alghamdi, Renate A. Schmidt, Dirk Walther and Yongsheng Gao

Published at: *Description Logics 2019*.

Abstract: Catering for ontology summary and reuse, several approaches such as modularisation and forgetting of symbols have been developed in order to provide users smaller sets of relevant axioms. We consider different module extraction techniques and show how they relate to each other. We also consider the notion of uniform interpolation that is underlying forgetting. We show that significant improvements in the performance of forgetting can be obtained by applying the forgetting tool to ontology modules instead of the entire ontology. We investigate combining several module notions with uniform interpolation and we provide a preliminary evaluation forgetting signatures based on the European Renal Association subset from SNOMED CT.

The program committee had the following members: Haniel Barbosa (co-chair), Giselle Reis (co-chair), Rob Blanco, Frédéric Blanqui, Simon Cruanes, Catherine Dubois, Amy Felty, Mathias Fleury, Stéphane Graham-Lengrand, Cezary Kaliszyk, Chantal Keller, Laura Kovács, Olivier Laurent, Stefan Mitsch, Carlos Olarte, Bruno Woltzenlogel Paleo, Florian Rabe, Martin Riener, Geoff Sutcliffe, Josef Urban and Yoni Zohar.

We would like to thank all authors for their submissions and all members of the program committee for the time and energy they spent to diligently ensure that accepted papers were of high quality. We also thank EasyChair for making it easy to chair the reviewing process. Furthermore, we are thankful to the CADE-27 organizer, Elaine Pimentel, for enabling a smooth local organization of the event.

In Natal we will have the honor to welcome two invited speakers: Assia Mahboubi, from Inria and Vrije Universiteit Amsterdam, giving a talk entitled *Systems for Doing Mathematics by Computer* and Thibault Gauthier, from Czech Technical University, presenting a talk entitled *Learning from Tactic Steps in Formal Proofs*.

The organization of this edition of PxTP stood on the shoulders of previous editions, and we are grateful to the chairs of previous editions for all the resources and infra-structure that they made available to us.

August 4, 2019

Giselle Reis and Haniel Barbosa

Table of Contents

Preface	i
<i>Giselle Reis and Haniel Barbosa</i>	
Table of Contents	iv
Converting ALC Connection Proofs into ALC Sequents	1
<i>Eunice Palmeira, Fred Freitas and Jens Otten</i>	
Invited Talk: Systems for Doing Mathematics by Computer	16
<i>Assia Mahboubi</i>	
Invited Talk: Learning from Tactic Steps in Formal Proofs	17
<i>Thibault Gauthier</i>	
Verifying Bit-vector Invertibility Conditions in Coq (Extended Abstract)	18
<i>Burak Ekici, Arjun Viswanathan, Yoni Zohar, Clark Barrett and Cesare Tinelli</i>	
EKSTRAKTO A tool to reconstruct Dedukti proofs from TSTP files (extended abstract)	27
<i>Mohamed Yacine El Haddad, Guillaume Burel and Frédéric Blanqui</i>	
Reconstructing veriT Proofs in Isabelle/HOL	36
<i>Mathias Fleury and Hans-Jörg Schurr</i>	
CLS-SMT: Bringing Together Combinatory Logic Synthesis and Satisfiability Modulo Theories ...	51
<i>Fadil Kallat, Tristan Schäfer and Anna Vasileva</i>	

Converting \mathcal{ALC} Connection Proofs into \mathcal{ALC} Sequents

Eunice Palmeira

Federal Institute of Alagoas
Maceió - AL, Brazil

eunicepalmeira@ifal.edu.br

Fred Freitas

Federal University of Pernambuco
Recife - PE, Brazil

fred@cin.ufpe.br

Jens Otten

University of Oslo
Oslo, Norway

jeotten@ifi.uio.no

The connection method has earned good reputation in the area of automated theorem proving, due to its simplicity, efficiency and rational use of memory. This method has been applied recently in automatic provers that reason over ontologies written in the description logic \mathcal{ALC} . However, proofs generated by connection calculi are difficult to understand. Proof readability is largely lost by the transformations to disjunctive normal form applied over the formulae to be proven. Such a proof model, albeit efficient, prevents inference systems based on it from effectively providing justifications and/or descriptions of the steps used in inferences. To address this problem, in this paper we propose a method for converting matricial proofs generated by the \mathcal{ALC} connection method to \mathcal{ALC} sequent proofs, which are much easier to understand, and whose translation to natural language is more straightforward. We also describe a calculus that accepts the input formula in a non-clausal \mathcal{ALC} format, what simplifies the translation.

1 Introduction

Description Logics (DLs) [1] are a family of knowledge representation formalisms considered as a fundamental foundation for the Semantic Web, as it constitutes the formalism underlying the Web Ontology Language (OWL) language. DL is an expressive, decidable subset of First Order Logic (FOL), successfully applied in several areas. DL provides a precise and unambiguous meaning to DL descriptions due to its formal semantics, and fast reasoners have been produced to the many fragments available [7].

One of them, the \mathcal{ALC} θ -Connections Calculus, and its automated reasoner RACCOON (Reasoner based on the Connection Calculus Over ONtologies), is based on the Connection Method [5, 8], and was specifically developed to infer over the Description Logic \mathcal{ALC} [5, 8]. The calculus includes typical DL features and techniques, such as notation without variables, absence of Skolem functions/unification and, inclusion of a blocking rule to handle cycles, which guarantees termination to make for the case of cyclic ontologies. The Connection Calculus has earned good reputation in the area of automated theorem proving due to its simplicity, efficiency and rational use of memory. The method represents formulae as matrices, whose columns are conjunctive clauses; its proof procedure consists of horizontally traversing paths through the matrix in order to connect complimentary literals (e.g., L with its complement $\neg L$). A pair $\{L, \neg L\}$, is called a connection, which corresponds to the validity the path being checked. Thus, a formula is valid if every path through the matrix corresponding to it has a connection.

Both calculi mentioned above, before attempting to find a proof, convert a formula into a disjunctive normal form. The translation to this clausal form often obscures the structure of the original formula and transforms some simple theorem proofs into difficult ones[11]. In complex cases, the deductions' premise(s) and conclusion can no longer be clearly identified, once the transformation has been applied [3]. Thus, proof readability and understandability is largely lost, and consequently, it becomes quite difficult to provide justifications and/or descriptions of the steps used during inferences.

The θ -Non-clausal \mathcal{ALC} θ -Connection Calculus is based on the \mathcal{ALC} θ -Connection Calculus and works directly on the structure of the original formula, thus avoiding the translation into a clausal form.

Nevertheless, its proof format is still not intuitive, once, like other connection calculi, it consists of a set of complementary pairs found in each path through the matrix, when the formula is valid.

The motivation of this work is to make a connection proof for \mathcal{ALC} more readable so that, in a near future, justifications can be generated automatically in natural language. Therefore, this article proposes a conversion method that translates non-clausal \mathcal{ALC} θ -connection proofs into \mathcal{ALC} sequent proofs. Sequent calculi have a more friendly proof representation than connection calculus; it conveys proofs in a formal logic argument style, where each proof line is a conditional tautology. Such translation should therefore contribute to a better user interaction with DL reasoners based on the Connection Method.

The DL \mathcal{ALC} is presented in the next section; Section 3 brings an \mathcal{ALC} non-clausal Connection Calculus for \mathcal{ALC} ; Section 4 introduces the \mathcal{ALC} Sequent Calculus, to which proofs will be translated; the conversion process and its main concepts in Section 5; an overview of the main algorithms for the conversion method with its computational complexities in Section 6; and conclusions in Section 7.

2 The Description Logic \mathcal{ALC}

An ontology O in \mathcal{ALC} is a set of axioms over a signature (N_C, N_R, N_O) , where N_C is the set of concept names (unary predicate symbols), N_R is the set of role or property names (binary predicate symbols); N_O is the set of individual names (constants) [1]. Concept expressions are inductively defined as follows. N_C includes \top , the universal concept that subsumes all concepts, and \perp , the bottom concept subsumed by all concept names belong to N_C . If $r \in N_R$ is a role and $C, D \in N_C$ are concepts, then the following formulae are also concepts: (i) $C \sqcap D$, (ii) $C \sqcup D$, (iii) $\neg C$, (iv) $\forall r.C$; (v) $\exists r.C$.

A knowledge base in DL consists of a set of basic axioms (TBox), and a set of axioms specific to a particular situation (ABox). Two axiom types are allowed in a TBox \mathcal{T} : (i) $C \sqsubseteq D$; (ii) $C \equiv D$, standing for $C \sqsubseteq D$ and $D \sqsubseteq C$. An ABox \mathcal{A} w.r.t. a TBox \mathcal{T} is a finite set of assertions of two types: (i) a concept assertion is a statement of the form $C(a)$, where $a \in N_O$, $C \in N_C$ and (ii) a role assertion $r(a, b)$, where $a, b \in N_O$, $r \in N_R$. An \mathcal{ALC} formula is either an axiom or an assertion; an ontology O is an ordered pair $(\mathcal{T}, \mathcal{A})$. The semantics of concepts and ontologies is defined in the usual way - see, e.g., [1].

3 The Non-clausal \mathcal{ALC} θ -Connection Calculus

Definition 1. (Query). A query $O \models \alpha$ is an \mathcal{ALC} formula to be proven valid, where O is an \mathcal{ALC} ontology, and α is either a TBox or an ABox axiom to be proven a logical consequence from O .

Definition 2. (Literal, clause, matrix). \mathcal{ALC} Literals are atomic concepts or roles, possibly negated or instantiated in the form L or $\neg L$. An \mathcal{ALC} disjunction is either a literal L , a disjunction $(E_0 \sqcup E_1)$ or an universal restriction $\forall r.E_0$. An \mathcal{ALC} conjunction is either a literal L , a conjunction $(E_0 \sqcap E_1)$ or an existential restriction $\exists r.E_0$, where E_0 and E_1 are expressions of arbitrary concepts (see DLs and its Mapping to FOL in [2]). Clauses are conjunctions of literals and matrices in the form $L_1 \sqcap \dots \sqcap L_m$, where each L_i is a literal or a matrix. A matrix of a formula (in DNF) is its representation as a set $\{C_1, \dots, C_n\}$, where each C_i is a clause.

Definition 3. (Formula with polarity). A formula with polarity, denoted by F^p , consists of a formula F and a polarity p , where $p \in \{0, 1\}$, that is, 0 is positive and 1 is negative. This concept is used to denote negation in a matrix, i.e. literals or matrices A and $\neg A$ are represented by A^0 and A^1 , respectively.

Definition 4. (\mathcal{ALC} Non-Clausal Matrix). An \mathcal{ALC} non-clausal matrix is a set of clauses in which a clause is a set of literals and matrices. Let F be a formula and p be a polarity. The matrix of F^p , denoted

by $M(F^P)$, is inductively defined according to Table 1, which indicates how the polarity is inherited by the (sub-)matrices of an F^P . The matrix of F^P is the matrix $M(F^0)$. Literals or (sub-)matrices involved in a universal restriction ($\forall r.C$) or in an existential restriction ($\exists r.C$) are underlined in the matrix.

Table 1: Matrix of an \mathcal{ALC} formula F^P .

Type	F^P	$M(F^P)$	Type	F^P	$M(F^P)$
Atomic	A^0	$\{\{A^0\}\}$	β	$(C \sqcap D)^0$	$\{\{M(C^0), M(D^0)\}\}$
	A^1	$\{\{A^1\}\}$		$(C \sqcup D)^1$	$\{\{M(C^1), M(D^1)\}\}$
α	$(\neg C)^0$	$M(C^1)$		$(C \sqsubseteq D)^1$	$\{\{M(C^0), M(D^1)\}\}$
	$(\neg C)^1$	$M(C^0)$	γ	$(\forall rD)^1$	$\{\{M(\underline{r^0}), M(\underline{D^1})\}\}$
	$(C \sqcap D)^1$	$\{\{M(C^1)\}, \{M(D^1)\}\}$		$(\exists rD)^0$	$\{\{M(\underline{r^0}), M(\underline{D^0})\}\}$
	$(C \sqcup D)^0$	$\{\{M(C^0)\}, \{M(D^0)\}\}$	δ	$(\forall rD)^0$	$\{\{M(\underline{r^1}), \{M(\underline{D^0})\}\}\}$
	$(C \sqsubseteq D)^0$	$\{\{M(C^1)\}, \{M(D^0)\}\}$		$(\exists rD)^1$	$\{\{M(\underline{r^1}), \{M(\underline{D^1})\}\}\}$
	$(C \models D)^0$	$\{\{M(C^1)\}, \{M(D^0)\}\}$			

Definition 5. (Positive) Graphical Representation of the Matrix. In the (positive) graphical representation of a matrix, its clauses are arranged horizontally, while the literals and (sub-)matrices of each clause are arranged vertically. The restrictions are represented by solid lines; when a restriction involves more than one clause, its literals are indexed in the bottom with the same index in the matrix column in the written representation, for example, the notation L_i (see example 1); restrictions with indexes are represented with horizontal lines; restrictions without indexes with vertical lines.

Example 1. (Query, clause, \mathcal{ALC} non-clausal matrix, formula with polarity, graphical representation of a matrix). The query $F_1 = \{\exists \text{hasPet.Cat} \sqsubseteq \text{CatOwner}, \text{OldLady} \sqsubseteq \exists \text{hasPet.Animal} \sqcap \forall \text{hasPet.Cat}\} \models \text{OldLady} \sqsubseteq \text{CatOwner}$ is read in FOL as:

$$\left. \begin{array}{l} \forall x((\exists y \text{hasPet}(x,y) \wedge \text{Cat}(y)) \rightarrow \text{CatOwner}(x)) \\ \forall z(\text{OldLady}(z) \rightarrow \exists v(\text{hasPet}(z,v) \wedge \text{Animal}(v))) \\ \wedge \forall k(\text{hasPet}(z,k) \rightarrow \text{Cat}(k)) \end{array} \right\} \models \forall u(\text{OldLady}(u) \rightarrow \text{CatOwner}(u))$$

and is represented by the FOL matrix (a is a Skolem terms, f a function symbol):

$$\{\{\text{hasPet}(x,y), \text{Cat}(y), \neg \text{CatOwner}(x)\}, \{\text{OldLady}(z), \{\neg \text{hasPet}(z, f(z)), \{\neg \text{Animal}(f(z)), \{\text{hasPet}(w,k), \neg \text{Cat}(k)\}\}, \{\neg \text{OldLady}(a)\}, \{\text{CatOwner}(a)\}\}\}$$

and by the following \mathcal{ALC} non-clausal matrix M_1 , which is defined according to 1 (column indices relate the two clauses involved in a same restriction; variables are omitted as they are specified implicitly):

$$\{\{\underline{\text{hasPet}^0}, \underline{\text{Cat}^0}, \text{CatOwner}^1\}, \{\text{OldLady}^0, \{\underline{\text{hasPet}^1}\}, \{\underline{\text{Animal}^1}\}, \{\underline{\text{hasPet}^0}, \underline{\text{Cat}^1}\}\}, \{\text{OldLady}(a)^1\}, \{\text{CatOwner}(a)^0\}\}$$

So, the graphical representation of M_1 is:

$$\left[\left[\begin{array}{c} \text{hasPet}^0 \\ \text{Cat}^0 \\ \text{CatOwner}^1 \end{array} \right] \left[\left[\begin{array}{c} \text{OldLady}^0 \\ \underline{[\text{hasPet}^1][\text{Animal}^1]} \\ \text{Cat}^1 \end{array} \right] \left[\begin{array}{c} \text{hasPet}^0 \\ \text{Cat}^1 \end{array} \right] \right] \left[\begin{array}{c} \text{OldLady}(a)^1 \\ \text{CatOwner}(a)^0 \end{array} \right] \right]$$

Matrices of the form $M = \{\dots, \{C_1, \dots, C_n\}, \dots\}$ can be simplified to $M' = \{\dots, C_1, \dots, C_n, \dots\}$, where C_1, \dots, C_n are clauses.

Clauses of the form $C = \{\dots, \{M_1, \dots, M_m\}, \dots\}$ can be simplified to $C' = \{\dots, M_1, \dots, M_m, \dots\}$, where M_1, \dots, M_m are matrices.

Definition 6. (Path). A *path* through a matrix $M = \{C_1, \dots, C_n\}$ is a set of literals containing a literal L_i of each clause $C_i \in M$, i.e., $\bigcup_{i=1}^n \{L_i\}$ with $L_i \in C_i$. A path through a matrix M (or a clause C) is inductively defined as follows. The (only) path through a literal L is $\{L\}$. If p_1, \dots, p_n are paths through the clauses C_1, \dots, C_n , respectively, then $p_1 \cup \dots \cup p_n$ is a path through the matrix $M = \{C_1, \dots, C_n\}$. If p_1, \dots, p_n are paths through the matrices/literals M_1, \dots, M_n , respectively, then p_1, \dots, p_n are also paths through the clause $C = \{M_1, \dots, M_n\}$.

Definition 7. (Connection, θ -substitution, θ -complementary connection). A *connection* is a pair of literals $\{E, \neg E\}$ with the same concept/role name, but different polarities. A *θ -substitution* assigns to each (possibly omitted) variable an individual or another variable (in the whole matrix). A *θ -complementary connection* is a pair of \mathcal{ALC} literals $\{E(x), \neg E(y)\}$ or $\{p(x, v), \neg p(y, u)\}$, with $\theta(x) = \theta(y), \theta(v) = \theta(u)$. The complement \bar{L} of a literal L is E if $L = \neg E$, and it is $\neg E$ if $L = E$.

Simple term unification without Skolem functions is used to calculate θ -substitutions. The application of a θ -substitution to a literal is an application to its variables, i.e. $\theta(E) = E(\theta(x))$ and $\theta(r) = r(\theta(x), \theta(y))$, where E is an atomic concept and r is a role. Furthermore, $x^\theta = \theta(x)$.

Example 2. (Path, Connection, θ -substitution, θ -complementary connection). In the matrix M_1 of Example 1, $\{\text{hasPet}^0 \mid, \underline{\text{hasPet}}_1^1, \underline{\text{Animal}}_1^1, \text{hasPet}^0 \mid, \text{OldLady}(a)^1, \text{CatOwner}(a)^0\}$ and $\{\text{Cat}^0, \underline{\text{hasPet}}_1^1, \underline{\text{Animal}}_1^1, \text{Cat}^1 \mid, \text{OldLady}(a)^1, \text{CatOwner}(a)^0\}$ are some paths through M_1 . $\{\text{Cat}^0 \mid, \text{Cat}^1\}$ is a connection. $\theta(\text{OldLady}^0) = \text{OldLady}(\theta(y))^0$ and $\theta(\text{hasPet}^0) = \text{hasPet}(\theta(y), x)^0$, where $\theta(y) = a$, are examples of θ -substitution, and $\{\text{OldLady}^0, \text{OldLady}(a)^1\}$ is a θ -complementary connection,

Definition 8. (Set of concepts, Skolem condition). The *set of concepts* $\tau(x)$ of a variable or individual x contains all concepts that were substituted/instantiated by x so far, i.e. $\tau(x) \stackrel{\text{def}}{=} \{E(x) \in \text{Path}\}$, where E is a concept and $E(x)$ is a substituted/instantiated literal coming from this concept. The *Skolem condition* ensures that at most one concept is underlined in the graphical matrix. The condition is formally stated as, $\forall a \mid \{E^i(a) \in \text{Path}\} \leq 1$, with a a variable/individual, and i a column index.

Definition 9. (α -Related Clause). Let C be a clause in a matrix M and L be a literal in M . C is α -related to L , iff M contains (or is equal to) a matrix $\{C_1, \dots, C_n\}$ such that $C = C_i$ or C_i contains C , and C_j contains L for some $1 \leq i, j \leq n$ with $i \neq j$. C is α -related clause to a set of literals \mathcal{L} , iff C is α -related to all literals $L \in \mathcal{L}$.

Example 3. (α -Related Clause) In the matrix of Example 1, $\{\underline{\text{Animal}}_1^1\}$ is α -related to $\{\text{hasPet}^0, \text{Cat}^1\}$.

Definition 10. (Parent Clause). Let M be a matrix and C be a clause in M . The clause $C' = \{M_1, \dots, M_n\}$ in M is called the *parent clause* of C iff $C \in M_i$ for some $1 \leq i \leq n$.

Example 4. (Parent Clause). In Example 1, $\{\text{OldLady}^0, \{\underline{\text{hasPet}}_1^1, \underline{\text{Animal}}_1^1, \{\text{hasPet}^0, \text{Cat}^1\}\}\}$ is parent clause of $\{\underline{\text{hasPet}}_1^1\}$.

Definition 11. (Extension Clause). Let M be a matrix and P a path (be a set of literals). Then the clause C in M is an *extension clause* of M with respect to P , iff either C contains a literal of P , or C is α -related to all literals of P occurring in M and if C has a parent clause, it contains a literal of P .

In the extension rule of the \mathcal{ALC} θ -Connection Calculus (3.1) the new subgoal clause (set of literals that need to be connected) is $C_2 \setminus \{L_2\}$. In the non-clausal connection calculus the extension clause C_2 might contain clauses that are α -related to L_2 and do not need to be considered for the new subgoal clause. Hence, these clauses can be deleted from the subgoal clause. The resulting clause is called the β -clause of C_2 with respect to L_2 .

Definition 12. (β -Clause). Let $C = \{M_1, \dots, M_n\}$ be a clause and L be a literal in C . The β -Clause of C with respect to L , denoted by $\beta\text{-Clause}_L(C)$, is inductively defined:

$$\beta\text{-Clause}_L(C) := \begin{cases} C \setminus \{L\} & \text{if } L \in C, \\ M_1, \dots, M_{i-1}, \{C^\beta\}, M_{i+1}, \dots, M_n & \text{otherwise,} \end{cases}$$

where $C' \in M_i$ contains L and $C^\beta := \beta\text{-Clause}_L(C')$.

Example 5. (Extension Clause, β -Clause). In Example 1, $C = \{\text{OldLady}^0, \{\text{hasPet}_1^1\}, \{\text{Animal}_1^1\}, \{\text{hasPet}^0, \text{Cat}^1\}\}$ is an extension clause with respect to $p = \{\text{CatOwner}(a)^0, \text{Cat}^0\}$, while the clause $\{\text{OldLady}^0, \{\text{hasPet}_1^1\}, \{\text{Animal}_1^1\}, \{\text{hasPet}^0\}\}$ is a β -Clause of C with respect to $L = \text{Cat}^1$.

3.1 The Formal Non-Clausal \mathcal{ALC} θ -Connection Calculus

Suppose we wish to entail if $O \models \alpha$ is valid using a direct method, like the Connection Method (CM). By the Deduction Theorem [3], we must then prove directly if $O \rightarrow \alpha$, or, in other words, if $\neg O \vee \alpha$ is valid. This opposes to classical refutation methods, like tableaux and resolution, which builds a proof by testing whether $O \cup \{\neg\alpha\} \models \perp$. Hence, in the CM, the whole knowledge base KB should be negated. Given $O = \{\alpha_1, \alpha_2, \dots, \alpha_n\}$, α_i being literal conjunctions in the clausal connection method, all (negated KB) formulae are converted to the Disjunctive Normal Form (DNF). A query then is the matrix $\neg O \vee \alpha$ (i.e., $\neg\alpha_1 \vee \neg\alpha_2 \vee \dots \vee \neg\alpha_n \vee \alpha$) to be proven valid. In the non-clausal calculus, instead of having clauses only with literals, they can also contain matrices, and no conversion is needed. If every path contains a (θ -complementary) connection (representing a subformula $A \sqcup \neg A$ in a disjunction, what makes this disjunction valid), then the matrix is valid.

Definition 13. (Non-Clausal \mathcal{ALC} θ -Connection Calculus) Figure 1 shows the rules of the formal non-clausal \mathcal{ALC} θ -connection calculus. Rules are applied bottom-up. The words of the calculus are tuples C, M, Path , where C is a clause, M is a matrix corresponding to query $O \models \alpha$ and Path is a set of literals. C is called the subgoal clause. C_1, C_2 and C_3 are clauses. The index $\mu \in \mathbb{N}$ of a clause C^μ denotes that C^μ is the μ -th copy of clause C , increased when Copy is applied for that clause (the variable x in C^μ is denoted x_μ). When Copy is used, it has to be followed by the application of Extension or Reduction, to avoid non-determinism in the rules application. The Blocking Condition is defined as follows: the new individual x_μ^θ (if it is new, then $x_\mu^\theta \notin N_O$, as in the condition) is only created if the set of concepts of the previously created individual $\tau(x_{\mu-1}^\theta)$ is not a subset of the set of concepts of the penultimate copied individual, i.e., $\tau(x_{\mu-1}^\theta) \not\subseteq \tau(x_{\mu-2}^\theta)$.

The calculus consists of six rules. The Axiom, Start, Reduction and Copy rules are the same as the ones from the \mathcal{ALC} θ -Connection Calculus. The Extension rule was modified to contain a β -Clause and the Decomposition rule [9] splits subgoal clauses into their sub-clauses.

Lemma 1. (Matrix characterization). A matrix M is valid iff there exist an index μ , a set of θ -substitutions $\langle \theta_i \rangle$ and a set of connections S , s.t. every path through M^μ , the matrix with copied clauses, contains a θ -complementary connection L_1^θ, L_2^θ in S , i.e. a connection with $\theta(L_1) = \theta(L_2)$. The tuple $\langle \mu, \langle \theta_i \rangle, S \rangle$ is called a matrix proof.

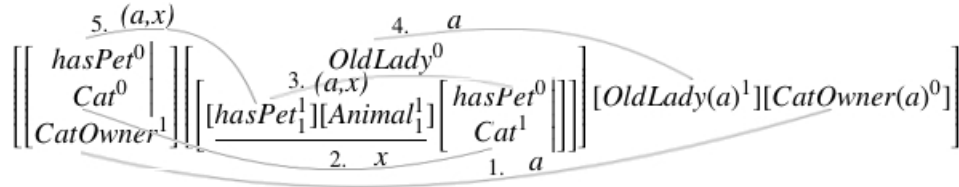
Example 6. (Non-Clausal \mathcal{ALC} θ -Connection Calculus). Figure 2 shows the proof for the F_1 of Example 1 using the matrix representation.

The proof starts (1) by choosing a clause from the consequent as the *start clause*, in this case, $\{\text{CatOwner}(a)\}$, and a literal of that clause is selected, $\text{CatOwner}(a)^0$. This literal is connected to

Axiom(A)	$\{\}, M, Path$
Start(S)	$\frac{C_1, M, \{\}}{\varepsilon, M, \varepsilon}$ with $C_1 \in \alpha$
Reduction(R)	$\frac{C, M, Path \cup \{L_2\}}{C \cup \{L_1\}, M, Path \cup \{L_2\}}$ with $\theta(L_1) = \theta(\overline{L_2})$ and the Skolem condition holds
Extension(E)	$\frac{C_3, M, Path \cup \{L_1\} \quad C, M, Path}{C \cup \{L_1\}, M, Path}$ with $C_3 := \beta\text{-clause}_{L_2}(C_2)$, C_2 is an extension clause of M wrt. $Path \cup \{L_1\}$, $L_2 \in C_2$, $\theta(L_1) = \theta(\overline{L_2})$ and the Skolem condition holds
Decomposition(D)	$\frac{C \cup C_1, M, Path}{C \cup \{M_1\}, M, Path}$ with $C_1 \in M_1$
Copy(C)	$\frac{C \cup \{L_1\}, M \cup \{C_2^\mu\}, Path}{C \cup \{L_1\}, M, Path}$ with C_2^μ is a copy of C_1 , $L_2 \in C_2^\mu$, $\theta(L_1) = \theta(\overline{L_2})$ and the blocking condition holds

Figure 1: Non-clausal \mathcal{ALC} θ -Connection Calculus.

$CatOwner^1$ by an extension step and instance a is the θ -substitution of $CatOwner^1$ and $CatOwner(a)^0$. This connection is still not enough to prove all the paths starting from $CatOwner(a)^0$; the paths that start in it and pass through the literals from the other connected clause, namely, Cat^0 and $hasPet^0$, are still to be verified. Indeed, each connection creates two sets of literals to be checked, the remaining literals from each of the clauses involved in the connection. In the new extension step (2), the connection $\{Cat^0, Cat^1\}$ is established on the variable (or fictitious individual) x , as it is not necessary yet to commit the substitution with an already existing individual. There is still remaining literals to be verified, the ones resulting from the clause to which Cat^0 belongs. Next (3), the $hasPet^0$ predicate is connected, and the θ -substitution generates the pair (y, x) (not shown in figure), for the connection. $OldLady^0$ is connected to $OldLady(a)^1$ (4), and then (5), when the connection $\{hasPet^0, hasPet^1\}$ is settled (using a reduction step, as there was already a connection with the same literal in the path), y was θ -substituted by x (i.e., $\theta(y) = x$), thus forming the pair (a, x) . This θ -substitution over y is then propagated through the path. Since every path through M_1 contains a θ -complementary connection, F_1 is valid. However, the readability of the proof is largely lost by the transformations applied on the formulas to be proven, making it difficult to translate the steps into natural language.

Figure 2: The \mathcal{ALC} non-clausal matrix proof of the F_1 using the graphical matrix representation.

Next, we present the Sequent Calculus to which \mathcal{ALC} non-clausal proofs will be translated.

4 An \mathcal{ALC} Sequent Calculus

According to [4], sequent calculi axiomatizes the relation of logical consequence (entailment), and this has an obvious parallel with the relation of subsumption, which is a keystone for DL representation and calculi. Bearing this in mind, Borgida et al proposed a sequent calculus for subsumption inferences in \mathcal{ALC} as an extension of the standard sequent calculus, in which there are no rules of implication, as they are indeed subsumption rules, so implication is replaced by \vdash without loss of meaning. In their calculus, terms are not moved from one side to the other of the turnstile during the proof, thus preserving the structure of the original subsumption, and in the case of multiple subsumptions, parentheses help in identifying the main subsumptions. Because of that, additional rules were created in which the negation is inserted in front of each construct, thus eliminating negation rules ($l\neg$, $r\neg$), what requires changing sequent antecedents to successors and vice versa. The calculus is divided in three parts: the first two describe sets of rules, while the last describes a set of axioms (see Figure 3, where a and b are arbitrary formulae and X and Y are arbitrary sequences of formulae).

- **Rules for propositional formulae:** rules \Box and \sqcup are duplicated by adding the negation rules for these connectives ($\neg\Box$, $\neg\sqcup$), while the proper negation rules (\neg) were modified to include the double negation rule ($\neg\neg$);
- **Rules for quantified formulae:** in [4], modal formulae are used ($r\Box$, $l\Diamond$) and their negated rules ($l\neg\Box$, $r\neg\Diamond$). Here, we replace these rules by their equivalents ($r\forall$, $l\exists$) and ($l\neg\forall$, $r\neg\exists$). The \exists -rules are the dual \forall -rules. A condition is explicitly considered for the application of these rules: the rule applies only if all homologous universal and existential formulae (e.g. $\forall h.C$ and $\exists h.C$ are homologous, $\forall h.C$ and $\exists f.C$ not) are joined together on the left and right sides of the sequent in the precondition. The rule is then applied only once;
- **Termination axioms:** unlike the standard sequent calculus, there are six termination axioms; all of them can be reduced to $X, a \vdash a, Y$ by applying the rules. The application of the \neg -rules forces formulae from the antecedent to the successor or vice versa, to be transformed until it gets to $X, a \vdash a, Y$, a procedure that is avoided in this calculus. Therefore, the additional termination axioms are necessary to ensure that formulae are never shifted from one side of the sequent to the other.

Although not stated explicitly, the calculus contains a cut rule, and the cut elimination theorem is valid in this case; it is stated below.

Theorem 1. Cut Elimination Theorem [6]. *Let S be a set of sequents (axioms) and s an individual sequent. $S \vdash_{SC} s$, if and only if, there is a proof in SC of s whose leaves are either logical or sequent axioms obtained by the substitution of S -belonging sequents, where the cut rule, $\frac{\Gamma \vdash \Delta, A \quad A, \Sigma \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}$, is only applied with a premise being an axiom.*

Example 7. (Sequent Proof for \mathcal{ALC} Subsumption). *Figure 4 shows Example 1's proof using the sequent calculus for \mathcal{ALC} . The cut rule is applied to the initial assumptions, according to theorem 1.*

This proof tree could be described by the following text in natural language: (1) If individuals who own at least one cat as a pet are owners of cats; and if the old ladies are, individuals who have at least one animal as a pet and all individuals who have only cat as pet. So this implies that old ladies own cats. (2) So, the old ladies are all people who have at least one cat as a pet. And all individuals who own at least one cat as a pet, own cats. (3) In addition to old ladies are all individuals who have at least one animal as a pet and all individuals who have only cat as pets; all individuals who have at least one animal as a pet and all individuals who have only cat as pets, are all individuals who have at least one cat as a pet. (4) Thus, an animal or a cat implies in a cat.

Rules for propositional formulae			
$\frac{X, a, b \vdash Y}{X, a \sqcap b \vdash Y}$	$(l\sqcap)$	$\frac{X \vdash a, Y \quad X \vdash b, Y}{X, \vdash a \sqcap b, Y}$	$(r\sqcap)$
$\frac{X, \neg a \vdash Y \quad X, \neg b \vdash Y}{X, \neg(a \sqcap b) \vdash Y}$	$(l\neg\sqcap)$	$\frac{X \vdash \neg a, \neg b, Y}{X \vdash \neg(a \sqcap b), Y}$	$(r\neg\sqcap)$
$\frac{X, a \vdash Y \quad X, b \vdash Y}{X, a \sqcup b \vdash Y}$	$(l\sqcup)$	$\frac{X \vdash a, b, Y}{X \vdash a \sqcup b, Y}$	$(r\sqcup)$
$\frac{X, \neg a, \neg b \vdash Y}{X, \neg(a \sqcup b) \vdash Y}$	$(l\neg\sqcup)$	$\frac{X \vdash \neg a, Y \quad X \vdash \neg b, Y}{X \vdash \neg(a \sqcup b), Y}$	$(r\neg\sqcup)$
$\frac{X, a \vdash Y}{X, \neg\neg a \vdash Y}$	$(l\neg\neg)$	$\frac{X \vdash a, Y}{X \vdash \neg\neg a, Y}$	$(r\neg\neg)$
Rules for quantified formulae			
$\frac{X' \vdash b, Y'}{X \vdash \forall r. b, Y}$	$(r\forall)$	$\frac{X', b \vdash Y'}{X, \exists r. b \vdash Y}$	$(l\exists)$
$\frac{X', \neg b \vdash Y'}{X, \neg\forall r. b \vdash Y}$	$(l\neg\forall)$	$\frac{X' \vdash \neg b, Y'}{X \vdash \neg\exists r. b, Y}$	$(r\neg\exists)$
where $X' = \{a \mid \forall r. a \in X\} \cup \{\neg a \mid \neg\exists r. a \in X\}$, and $Y' = \{a \mid \exists r. a \in Y\} \cup \{\neg a \mid \neg\forall r. a \in Y\}$			
Termination axioms			
$X, a \vdash a, Y$	$(=)$	$X, \neg a \vdash \neg a, Y$	$(=)$
$X, a, \neg a \vdash Y$	$(l\uparrow)$	$X \vdash a, \neg a, Y$	$(r\uparrow)$
$X, \perp \vdash Y$	$(l\perp)$	$X \vdash \top, Y$	$(l\top)$
Cut rule			
$\frac{\Gamma \vdash \Delta, A \quad A, \Sigma \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}$			

Figure 3: The Sequent Calculus for \mathcal{ALC} Subsumption [4].

$$\begin{array}{c}
 \frac{\text{TRUE}}{A, C \vdash C} = \\
 \frac{\frac{\frac{\frac{OL \vdash \exists h. A \sqcap \forall h. C}{\exists h. A, \forall h. C \vdash \exists h. C} l\exists}{\exists h. A \sqcap \forall h. C \vdash \exists h. C} l\sqcap}{\exists h. A \sqcap \forall h. C \vdash \exists h. C} \text{cut}}{\frac{OL \vdash \exists h. C \quad \exists h. C \vdash CO}{(\exists h. C \vdash CO, OL \vdash \exists h. A \sqcap \forall h. C) \vdash (OL \vdash CO)} \text{cut}} \\
 \frac{\quad}{((\exists h. C \vdash CO) \sqcap (OL \vdash \exists h. A \sqcap \forall h. C)) \vdash (OL \vdash CO)} l\sqcap
 \end{array}$$

Figure 4: \mathcal{ALC} sequent proof for F_1 . The names of the clauses and the roles are abbreviated.

5 Conversion Method

The process consists of two steps: building a formula tree and then converting this formula tree into sequents, given an \mathcal{ALC} query and its matrix non-clausal connection proof. They are explained below.

5.1 Building the Formula Tree

Definition 14. (Formula Tree, Position, Label, Polarity, Type). A *formula tree* is a syntactic representation of a formula F as a tree, where each node can have up to two child nodes. Each node has:

Position: an index that identifies each element (predicate or connective) in the formula. Its represented as a_0, a_1, a_2, \dots ; **Label:** either a connective ($\sqcap, \sqcup, \neg, \exists, \forall$), quantifier or predicate, if it is an atomic

(sub-)formula. Nodes whose label is a predicate are leaves of the tree (figure 5b), while other nodes are internal (figure 5a); **Polarity**: can be 0 or 1. It is determined by the label and the parent node polarity. The root node of the tree has polarity 0; **Type**: the type of a node is a Greek letter: α , β , α' , β' , γ and δ . It is determined by its label and its polarity. Leaf nodes have no type. The polarity and type of a node are defined in table 2. For example, in the first line of this table, $(A \sqcap B)^1$ means that the node labelled \sqcap and polarity 1 has type α and its successor nodes have polarity 1.

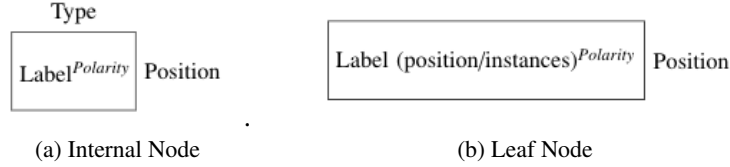


Figure 5: Node Representation.

Table 2: Polarity and types of nodes for \mathcal{ALC}

Type α	$(A \sqcap B)^1$	$A^1 \quad B^1$	Type β	$(A \sqcap B)^0$	$A^0 \quad B^0$	Type δ	$(\forall rA)^0$	$r^1 \quad A^0$
	$(A \sqcup B)^0$	$A^0 \quad B^0$		$(A \sqcup B)^1$	$A^1 \quad B^1$		$(\exists rA)^1$	$r^1 \quad A^1$
	$(\neg A)^1$	A^0						
	$(\neg A)^0$	A^1						
Type α'	$(A \sqsubseteq B)^0$	$A^1 \quad B^0$	Type β'	$(A \sqsubseteq B)^1$	$A^0 \quad B^1$	Type γ	$(\forall rA)^1$	$r^0 \quad A^1$
	$(A \models B)^0$	$A^1 \quad B^0$					$(\exists rA)^0$	$r^0 \quad A^0$

Nodes of type α and α' correspond to sequent rules that do not cause proof branching. Nodes of type γ and δ correspond to quantifier rules. Rules associated to type δ have the eigenvariable condition in the sequent calculi (where the term t , the eigenvariable in the inference, appears in the main formula of inference and in no other formula in the sequent. In the case of the \exists rule for the existential quantifier and $r\forall$ rule for the universal quantifier). Nodes of type β and β' (i.e., \sqcap^0 , \sqcup^1 , and \sqsubseteq^1) are particularly important, since their respective rules in sequents (described in table 3) split proof branching into two independent sub-proofs. Nodes have their types indexed in the formula tree to facilitate their identification, for example $\beta_1, \beta_2, \beta'_1, \beta'_2$. Each branch whose root is of type β or β' is marked with a letter (a,b,c,...).

Leaf nodes with instances are children of nodes type α , α' or β . Leaf nodes without instances have labels attached to their closest predecessor nodes' position, according to the following criteria : (1) if the leaf node label represents a concept, it has an unique position associated to its label; (2) if the leaf node label represents a role, it has two positions associated to its label in the form (a_1, a_2) , where a_2 is the of the nearest predecessor node's position; (3) only type γ , δ and β' node positions are associated to the labels. This helps to check for complementarity in a connection between two nodes.

The tree construction is guided by the identification of the (sub-)formulae's main constructor (connective or quantifier), which will be a label in the tree node. This node has at most two branches that binds them to their child nodes, i.e., new (sub-)formulas. The node type and its childrens polarities are assigned according to table 2. If children nodes are not atomic (sub-)formulae, the process repeats itself by identifying these (sub-)formulae's main constructor and then generating other nodes in the tree, until it reaches the leaves.

The proof matrix elements must correspond to the leaf nodes in the formula tree, indicated by the position of the corresponding predicate, as explained in section 5.2 step 2.

Example 8. (Building the Formula Tree Process). Figure 6 shows the first step in the tree construction for F_1 from Example 1: $((\exists h.C \sqsubseteq CO) \sqcap (OL \sqsubseteq \exists h.A \sqcap \forall h.C)) \models (OL(a) \sqsubseteq CO(a))$. Its main constructor is \models , the root node label, which, by definition has polarity 0; its position is a_0 . According to table 2, its type is α' ; its children nodes, on the right and left, have polarities 0 and 1, respectively, and both are sub-formulas of \models in F_1 . This process continues until it reaches the leaf nodes, as shown in figure 7.

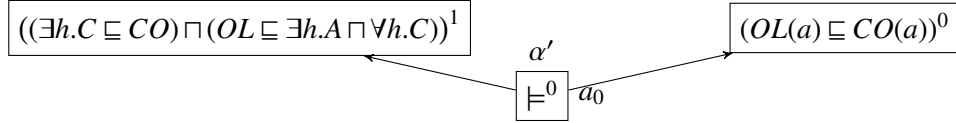


Figure 6: Step 01 Process of building the formula tree for F_1 .

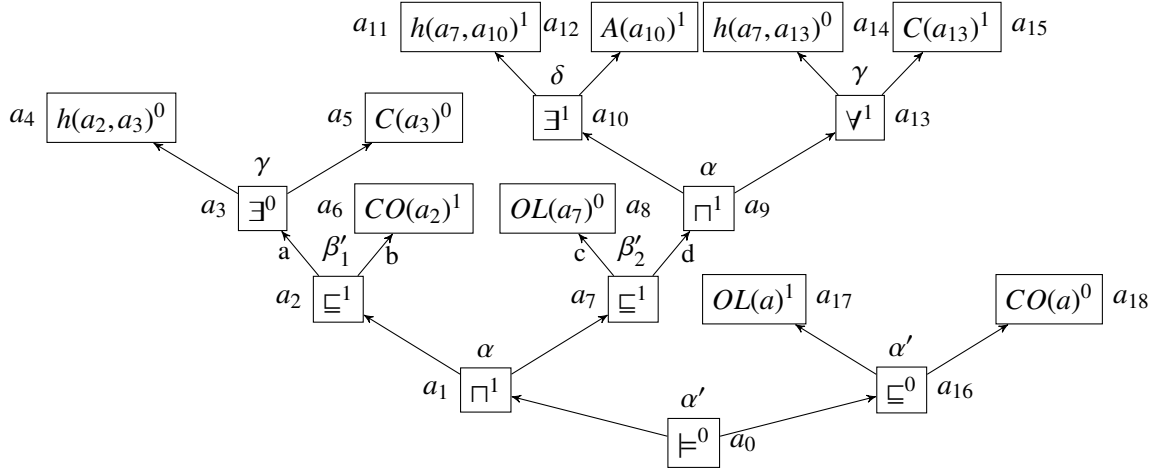


Figure 7: Formula Tree for F_1 with labels, polarities and types.

For a given formula A , A' , B , B' , Γ and Δ are used to denote the sets of node positions of type α , α' , β , β' , γ , and δ , respectively.

Definition 15. (Substitution of positions σ_δ , ordering relation \sqsubseteq_δ). It replaces positions of type γ for positions of type δ . A **position substitution** σ_δ is a mapping of the set Γ of type γ node positions to the set Δ of type δ node positions. The σ_δ substitution induces a **partial ordering relation** \sqsubseteq_δ in $\Delta \times \Gamma$ as follows: let $u \in \Gamma$ and $v \in \Delta$; if $\sigma_\delta(u) = p$ then $v \sqsubseteq_\delta u$ for all $v \in \Delta$ occurring in position p .

Since the sequent rules $r\forall$ and $l\exists$ and their homologues $l\neg\forall$ and $r\neg\exists$ are restricted to the eigenvariable condition, the relation $v \sqsubseteq_\delta u$ expresses that the node labelled by v must be reduced before reducing the one labelled by u .

Example 9. (Substitution of positions σ_δ , ordering relation \sqsubseteq_δ). Consider the formula tree in figure 7. Let u be the node labelled by \forall^1 , with position a_{13} and type γ , and let v be the node labelled by \exists^1 , with position a_{10} and type δ . To replace the position of a Type γ node by the position of a type δ node, It is necessary to reduce the type δ node first, then the node with the position a_{10} must be reduced before the node with the position a_{13} . Thus, for this example, the ordering relation \sqsubseteq_δ is given by $\exists^1 a_{10} \sqsubseteq_\delta \forall^1 a_{13}$, and the substitution $\sigma_\delta(\forall^1 a_{13}) = a_{10}$. With this, we have $\sigma_\delta = \{a_{13}/a_{10}\}$.

Definition 16. (Substitution of positions $\sigma_{\beta'}$). It replaces positions of type β' , γ , δ for instances or positions of type β' . Positions of the nodes of type β' , γ and δ , as well as instances, appear in atomic formulas, so a **substitution of positions** $\sigma_{\beta'}$ is a mapping of the set $B' / \Gamma / \Delta$ positions of nodes of type $\beta' / \gamma / \delta$ to instances or positions of nodes of type β' . Let u be a leaf node with the positions of nodes of type $\beta' / \gamma / \delta$ associated to its label and $v \in B'$; if $\sigma_{\beta'}(u) = p$, where $p \in B'$ or p is an instance.

Reducing a node means applying the sequent rule that corresponds to that node over a given (sub-)formula. Leaf nodes are not reduced.

Example 10. (Substitution of positions $\sigma_{\beta'}$). Consider the formula tree in Figure 7. Let u be the node labelled by OL^0 , with position a_8 and position a_7 of type β' associated to its label, and let v be the node labelled by OL^1 , with position a_{17} and instance a . The substitution for this in leaf u in this case is $\sigma_{\beta'}(OL(a_7)^0) = a$. Therefore, $\sigma_{\beta'} = \{a_7/a\}$.

Definition 17. (Substitution σ_{Final}). It is a combination of σ_{δ} and $\sigma_{\beta'}$. A σ_{Final} **substitution** consists of a substitution σ_{δ} and a substitution $\sigma_{\beta'}$, where $\sigma_{Final} := \sigma_{\delta} \cup \sigma_{\beta'}$.

Example 11. (Substitution σ_{Final}). Considering the two previous examples, $\sigma_{Final} = \{a_{13}/a_{10}, a_7/a\}$.

Definition 18. (Connection, σ_{Final} -complementary connection). A **connection** is a pair of leaf nodes labelled with the same predicate symbol and the same position associated with the label or the same instance, but with different polarities. If they are identical under σ_{Final} , the connection is a σ_{Final} -**complementary connection**.

Example 12. (Connection, σ_{Final} -complementary connection). Let the formula tree in figure 7 be. The leaf nodes $h(a_2, a_3)^0$ and $h(a_7, a_{10})^1$ with positions a_4 and a_{11} , respectively, form a connection that is complementary under $\sigma_{Final} = \{a_2/a_7, a_3/a_{10}\}$.

Definition 19. (Tree Ordering $<$). The **tree ordering** $<$ of an F formula is the partial ordering of the nodes positions in the tree formula. $<$ is defined as follows: (i) the root occupies the smallest position with respect to this ordering, (ii) $a_i < a_j$ if and only if the position a_i is below a_j in the formula tree.

Example 13. (Tree Ordering $<$). In the tree from Figure 7, there are examples of tree ordering: $a_7 < a_9 < a_{13} < a_{15}$ and $a_0 < a_1 < a_2 < a_3$.

Definition 20. (Reduction Order \triangleleft). The transitive closure of the union of \sqsubset_{δ} , $\sqsubset_{\beta'}$ and $<$ is called **reduction order** \triangleleft , i.e., $\triangleleft := (< \cup \sqsubset_{\delta} \cup \sqsubset_{\beta'})^+$.

Nodes $v \triangleleft u$ means that the node v must be reduced before the node labelled by u in the sequent proof. \triangleleft determines the nodes' reduction order, and helps determine which sequent rules are to be used and in which order.

Example 14. (Reduction Order \triangleleft). In Figure 7, the nodes with positions a_7 , a_{10} , a_{16} and a_{13} , have the following reduction order \triangleleft : (i) $a_7 \triangleleft a_{10}$; (ii) $a_7 \triangleleft a_{13}$; (iii) $a_{10} \sqsubset_{\delta} a_{13}$. The orderings' union and the tree ordering determine the reduction order for these nodes: $a_7 \triangleleft a_{10} \triangleleft a_{13}$.

Definition 21. (σ_{Final} Admissible Substitution). An σ_{Final} **Substitution is admissible** if the reduction order \triangleleft is not reflexive. In this case, it is possible to construct a sequent proof.

A correspondence between node label, polarity and type with the sequent rules presented in section 4, is established in table 3. Such correspondence is useful for the sequent proof construction, where the polarity helps in the identification of the rule. Polarity 1 represents a rule on the left (left or l); polarity 0, on the right (right or r), for cases where there is already an associated rule. For instance, in Table 3's first line, for node \sqcap^1 the rule is $l\sqcap$, while for node \sqcap^0 it is $r\sqcap$. For cases where internal nodes are preceded by a node labelled by a negation, correspondences are in Table 3's last four columns.

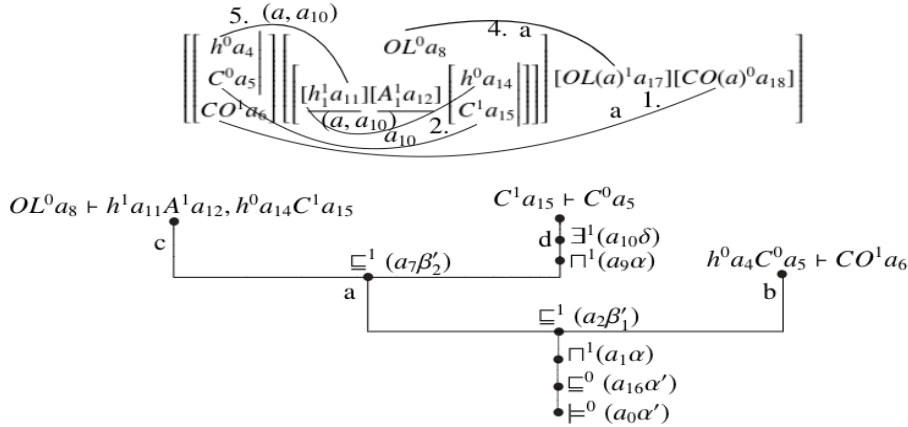
Table 3: Correspondence between label, polarity and type of a node, preceded or not by a node labelled with negation, to \mathcal{ALC} Sequent rules.

Not preceded						Preceded			
Type α	Rule	Type β	Rule	Type δ	Rule	Type α	Rule	Type β	Rule
\sqcap^1	$l\sqcap$	\sqcap^0	$r\sqcap$	\forall^0	$r\forall$	\neg^1	$r\neg\neg$	\sqcap^0	$l\neg\sqcap$
\sqcup^0	$r\sqcup$	\sqcup^1	$l\sqcup$	\exists^1	$l\exists$	\neg^0	$l\neg\neg$	\sqcup^1	$r\neg\sqcup$
\neg^1	\emptyset					\sqcap^1	$r\neg\sqcap$		
\neg^0	\emptyset					\sqcup^0	$l\neg\sqcup$		
Type α'	Rule	Type β'	Rule	Type γ	Rule			Type δ	Rule
\sqsubseteq^0	\emptyset	\sqsubseteq^1	Cut	\forall^1	\emptyset			\forall^0	$l\neg\forall$
\vDash^0	\emptyset			\exists^0	\emptyset			\exists^1	$r\neg\exists$

5.2 Conversion to Sequents

Given an \mathcal{ALC} query and its matricial non-clausal connection proof, the conversion procedure transforms this proof into an \mathcal{ALC} sequent proof. This process performs four steps, which are described below:

- **Step 1- Formula tree construction:** A syntactic representation in tree form is constructed for the input formula, containing nodes, as described in 14. The position of each predicate is input to step 2, and the tree to steps 3 and 4. **Example:** The conversion process begins with the F_1 formula tree construction, described in definition 14, which resulted in the formula tree represented in figure 7.
- **Step 2- Matrix elements' positions assignment:** Since proof matrix elements correspond to predicates in the formula and also to leaf nodes in the formula tree, this step assigns to each matrix element the position of the corresponding predicate. Its input is the matrix non-clausal connection proof and the position of predicates. Its output is input to step 3. **Example:** Each element of the matrix is assigned with the position of the corresponding predicate in the formula, see matrix in 8.

Figure 8: Steps representation in the connection proof/sequent \mathcal{ALC} for F_1 .

- **Step 3- (partial) sequent proof structure Construction:** The matrix non-clausal connection proof with the positions of each element and the formula tree are inputs for this step. To each matrix connection, the formula tree is examined in search for the leaf nodes that correspond to the connection. The paths between the root node and these nodes in the tree are analyzed to determine the order of nodes to be worked on and thus build a structure of the (partial) proof in sequents. This

structure provides information about the reduction order \triangleleft , which helps determine the rules to be applied, and on the existence of the proof branch, given by the identification of the nodes of type β and β' . The (partial) sequent proof structure constructed will be the input for step 4. **Example:** The first connection links element $CO(a)^0$, from position a_{18} , to element CO^1 , of position a_6 , which are complementary under the substitution $\sigma_{\beta'} = \{a_2/a\}$, see table 4. The path between these leaf nodes is $\{a_{18}, a_{16}, a_0, a_1, a_2, a_6\}$. Since there is no ordering relation \sqsubset_σ between the nodes of that path and there are two tree orderings given by $a_0 < a_{16}$ and $a_0 < a_1 < a_2$, It is possible to start with any of these tree orderings. Choosing the first, we have the order of reduction at that moment equal to: $a_0 \triangleleft a_{16} \triangleleft a_1 \triangleleft a_2$. Since the node with position a_2 is of type β' , the sequent is divided into two branches, called a e b, as in the formula tree. Thus, this connection closes the branch b, branch where the node CO^1 is, and leads to the axiom $h^0, C^0 \vdash CO^1$, because nodes of type β' are associated with the cut rule (see table 3). In the second connection, \underline{C}^0 , with position a_5 in branch a, is connected to \underline{C}^1 , with position a_{15} in branch d, and the path between them is $\{a_5, a_3, a_2, a_1, a_7, a_9, a_{13}, a_{15}\}$. As the nodes with positions a_1 and a_2 have already been reduced, it is necessary to reduce the nodes with positions, a_3, a_7, a_9 and a_{13} , which have tree ordering $a_7 < a_9 < a_{13}$ and the relations $a_{10} \sqsubset_\delta a_3$ and $a_{10} \sqsubset_\delta a_{13}$. At the moment it is only possible to reduce the node with position a_7 and then the node with position a_9 , that is, $a_7 \triangleleft a_9$. Since the node with position a_7 is of type β' , its reduction divides branch 'a' into branches 'c' and 'd'. Then the node with position a_9 , in branch 'd', is reduced. Since there are pendant nodes on this path, it is not yet possible to form an axiom and close the 'd' branch. The third connection is analyzed, where \underline{h}^0 , with position a_{14} , is connected to \underline{h}^1 , with position a_{11} , both in branch 'd'. The path between the nodes is $\{a_{14}, a_{13}, a_9, a_{10}, a_{11}\}$. Since a_9 has already been reduced, and there are the relations $a_{10} \sqsubset_\delta a_{13}$ and $a_{10} \sqsubset_\delta a_3$, the a_{10} position node is reduced, and 'together' with it the nodes with position a_{13} and a_3 . The reduction of the a_{10} position node makes the third and second connection complementary under the substitutions $\sigma_\delta = \{a_{13}/a_{10}, a_3/a_{10}\}$. With this the last two connections are reflected in the sequent proof leading to the closure of the 'd' branch. Notice that the second connection was only reached in the tree after the third connection, this leads to the axiom in the form $C^1 \vdash C^0$. The fourth connection connects OL^0 , with position a_8 in branch 'c', to OL^1 , with position a_{17} . The path between the nodes with theses positions is $\{a_8, a_7, a_1, a_0, a_{16}, a_{17}\}$. As all nodes on this path have already been reduced, no reduction will be necessary in this step. Thus, 'c' branch is closed with an axiom in the form $OL^0 \vdash h^1 A^1, h^0 C^1$, due to the cut rule. This connection is complementary under $\sigma_{\beta'} = \{a_7/a\}$. On the fifth and last connection, which connects \underline{h}^0 to \underline{h}^1 , there is no need of node reduction, since all nodes in the path were reduced. The connection is complementary under $\sigma_\delta = \{a_3/a_{10}\}$. Note that a_2/a and a_7/a were $\sigma_{\beta'}$ previous substitutions. All connections are complementary under a substitution σ_{Final} , all branches of the proof structure in sequent were closed, and the reduction order is not reflexive, as shown in figure 8 and in Table 4.

- **Step 4- Construction of the complete sequent proof:** Here, the process builds a complete sequent proof (output) from the (partial) sequent proof structure and the correspondence between nodes and sequent rules, described in 3. The input is (partial) sequent proof structure, the formula tree and \mathcal{ALC} sequent rules. **Example:** The structure obtained in step 3 is traversed. The proof begins with the reduction of a_1 position node, since the first two tree nodes do not have associated rule, because they are of type α' . Rule \sqcap is applied. Then, the a_2 position node, with type β' , reduced by means of the cut rule on the query α , that is, on $(OL \vdash CO)$. The proof is divided into branches 'a' and 'b'. The 'b' branch is closed with the initial axiom $\exists h.C \vdash CO$, while branch 'a' is open, in which $OL \vdash \exists h.C$ must be proved. The next node is of position a_7 , of type β' , and its reduction

Table 4: Relation between connections, substitutions and orderings

N	Nodes	σ_δ	$\sigma_{\beta'}$	\sqsubset_δ	\triangleleft
1	$CO(a_2)^1 a_6, CO(a)^0 a_{18}$		a_2/a		$a_0 \triangleleft a_{16} \triangleleft a_1 \triangleleft a_2$
2	$C(a_3)^0 a_5, C(a_{13})^1 a_{15}$	$a_{13}/a_{10},$ a_3/a_{10}		$a_{10} \sqsubset_\delta a_3,$ $a_{10} \sqsubset_\delta a_{13}$	$a_7 \triangleleft a_9$
3	$h(a_7, a_{13})^0 a_{14}, h(a_7, a_{10})^1 a_{11}$	a_{13}/a_{10}			a_{10}
4	$OL(a_7)^0 a_8, OL(a)^1 a_{17}$		a_7/a		
5	$h(a_2, a_3)^0 a_4, h(a_7, a_{10})^1 a_{11}$	a_3/a_{10}	a_2/a		
$\sigma_{Final} = a_2/a, a_{13}/a_{10}, a_3/a_{10}, a_7/a$			$a_0 \triangleleft a_{16} \triangleleft a_1 \triangleleft a_2 \triangleleft a_7 \triangleleft a_9 \triangleleft a_{10}$		

divides the branch 'a' into branches 'c' and 'd', by means of the application of a new cut rule on $OL \vdash \exists h.C$. The 'c' branch is closed with the initial axiom $OL \vdash \exists h.A \sqcap \forall h.C$, while the 'd' branch stays open. To close the 'd' branch, the a_9 position node is reduced with the $l\sqcap$ rule, followed by the node with position a_{10} , through rule $l\exists$. This ends the F_1 sequent proof, as shown in figure 9:

$$\begin{array}{c}
 \frac{\frac{\frac{A, C \vdash C}{\exists h.A, \forall h.C \vdash \exists h.C} \exists \text{I}}{\exists h.A \sqcap \forall h.C \vdash \exists h.C} l\sqcap}{OL \vdash \exists h.A \sqcap \forall h.C} \text{cut}}{\frac{OL \vdash \exists h.C}{(\exists h.C \vdash CO, OL \vdash \exists h.A \sqcap \forall h.C) \vdash (OL \vdash CO)} \text{cut}} \frac{\exists h.C \vdash CO}{((\exists h.C \vdash CO) \sqcap (OL \vdash \exists h.A \sqcap \forall h.C)) \vdash (OL \vdash CO)} l\sqcap
 \end{array}$$

Figure 9: Complete proof in \mathcal{ALC} sequents for F_1 .

6 Complexity

This section presents a very brief overview of the main algorithms for the conversion method with its complexities, according to the 4 steps seen in section 5.2. All the algorithms are demonstrated in [10]. Time complexities were analyzed according to the input size of each algorithm. For example, some algorithms receive an \mathcal{ALC} formula F as input, so the input size n represents the number of symbols of F . Other algorithms accept an F proof matrix as input; in this case, the input size is the matrix number of symbols, including connections between literals. This input is represented by m .

Figure 10 presents the main algorithms' execution order. Lines with arrows indicate that the output of one algorithm is input to another. For example, the output from algorithm 02 (called *convertsPostFix*) is conveyed as input for algorithm 03 (called *buildTree* and 04 (called *assignPosition*). The complexity of algorithm 05 (*Search Connections*) is the highest among the algorithms: $O(n^4)$, up to four iterations over structures based on the input size m .

7 Conclusions

This work presents a method to convert Non-clausal \mathcal{ALC} connections proofs into more readable proofs. The approach consists in transforming these proofs into proofs in the \mathcal{ALC} -Sequent Calculus [4]. Hence,

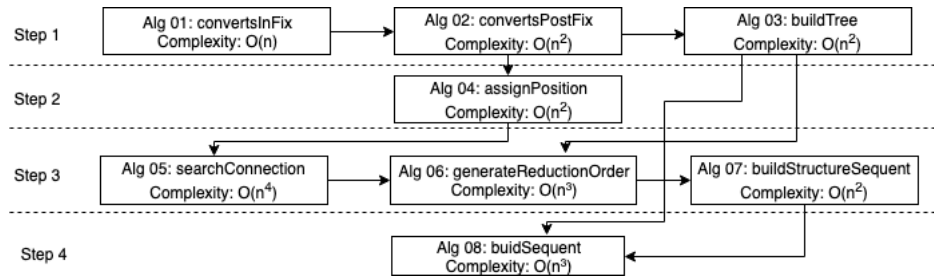


Figure 10: Overview of the main algorithms' order.

this conversion assumes that the input formulae will always be in non-clausal form, i.e., without the need to transform these formulae into any normal form. A tree representation of formulae is used as a guide in this conversion and a sequent proof is created while the connection proof is traversed. This conversion must contribute to describe how the reasoners based on the \mathcal{ALC} Connection Method summon their inferences and may facilitate the creation of natural language explanations, given the ease of converting sequents to texts. The evaluation of the main algorithms' computational complexities demonstrates its practical feasibility, since they display polynomial complexity. In this perspective, the scientific contributions of this work should characterize the importance of the logical proofs, clarify the reasoning process and increase inferences' readability, thus providing better user interaction with connection reasoners.

References

- [1] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi & P. F. Patel-Schneider, editors (2003): *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.
- [2] F. Baader, I. Horrocks & U. Sattler (2008): *Description Logics*. In: *Handbook of Knowledge Representation, Foundations of Artificial Intelligence 3*, Elsevier, pp. 135–179, doi:10.1016/S1574-6526(07)03003-9.
- [3] W. Bibel (1993): *Deduction - automated logic*. Academic Press.
- [4] A. Borgida, E. Franconi & I. Horrocks (2000): *Explaining \mathcal{ALC} Subsumption*. In: *ECAI 2000, Proceedings of the 14th European Conference on Artificial Intelligence, Berlin, Germany, 2000*, pp. 209–213.
- [5] F. Freitas & J. Otten (2016): *A Connection Calculus for the Description Logic \mathcal{ALC}* . In: *Advances in Artificial Intelligence - 29th Canadian Conference on Artificial Intelligence, Canadian AI 2016, Victoria, BC, Canada, May 31 - June 3, 2016. Proceedings*, pp. 243–256, doi:10.1007/978-3-319-34111-8_30.
- [6] Jean-Yves Girard, Paul Taylor & Yves Lafont (1989): *Proofs and Types*. Cambridge University Press.
- [7] I. Horrocks (2008): *Ontologies and the semantic web*. *Commun. ACM* 51(12), pp. 58–67, doi:10.1145/1409360.1409377.
- [8] D. Melo, F. Freitas & J. Otten (2017): *RACCOON: A Connection Reasoner for the Description Logic \mathcal{ALC}* . In: *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, pp. 200–211.
- [9] J. Otten (2011): *A Non-clausal Connection Calculus*. In: *Automated Reasoning with Analytic Tableaux and Related Methods - 20th International Conference, TABLEAUX 2011, Bern, Switzerland, July 4-8, 2011. Proceedings*, pp. 226–241, doi:10.1007/978-3-642-22119-4_18.
- [10] E. Palmeira (2017): *Conversion of Proof in Description Logic \mathcal{ALC} Generated by Connection Method into Sequents*. Ph.D. thesis, Federal University of Pernambuco.
- [11] D. A. Plaisted & S. Greenbaum (1986): *A Structure-Preserving Clause Form Translation*. *J. Symb. Comput.* 2(3), pp. 293–304, doi:10.1016/S0747-7171(86)80028-1.

Systems for Doing Mathematics by Computer

Assia Mahboubi

Inria
France

Vrije Universiteit Amsterdam
The Netherlands

Assia.Mahboubi@inria.fr

In the late 80s, Wolfram describes the Mathematica computer algebra system as a *system for doing mathematics by computer* [6]. Since, a broad spectrum of tools have been designed for computer-aided mathematics: computer algebra systems offer sophisticated algorithms for symbolic computations; scientific computing builds on the implementation of powerful numerical analysis algorithms; high-performance automated provers have produced proofs for long-standing mathematical conjectures [4]; proof assistants are mature enough for the formalization of contemporary mathematics [5, 2], etc. The ever eased access to the computational power of machines has changed the face of experimentation in mathematics.

But the status of proofs in the mathematical literature has been transformed as well, as these can themselves be computer-aided [3, 5]. Proof assistants provide both the most expressive language to represent mathematical objects, and the highest possible guarantee on the correctness of formalized proofs. They could thus in principle be used to organize a fruitful cooperation among all these systems. But delicate software engineering problems, as well as more fundamental translation issues often hinder this collaboration [1]. In this talk we will try to discuss and illustrate the different views these different software may have on various mathematical objects, in particular from the perspective of proof exchange and verification.

References

- [1] Frédéric Chyzak, Assia Mahboubi, Thomas Sibut-Pinote & Enrico Tassi (2014): *A Computer-Algebra-Based Formal Proof of the Irrationality of $\zeta(3)$* . In Gerwin Klein & Ruben Gamboa, editors: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings, Lecture Notes in Computer Science 8558*, Springer, pp. 160–176, doi:10.1007/978-3-319-08970-6_11.
- [2] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O’Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi & Laurent Théry (2013): *A Machine-Checked Proof of the Odd Order Theorem*. In Sandrine Blazy, Christine Paulin-Mohring & David Pichardie, editors: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings, Lecture Notes in Computer Science 7998*, Springer, pp. 163–179, doi:10.1007/978-3-642-39634-2_14.
- [3] Thomas C. Hales (2005): *A proof of the Kepler conjecture*. *Annals of Mathematics* 162(3), pp. 1065–1185, doi:10.4007/annals.2005.162.1065.
- [4] Marijn J. H. Heule (2018): *Schur Number Five*. In: *AAAI*, AAAI Press, pp. 6598–6606.
- [5] Fabian Immler (2018): *A Verified ODE Solver and the Lorenz Attractor*. *J. Autom. Reasoning* 61(1-4), pp. 73–111, doi:10.1007/s10817-017-9448-y.
- [6] Stephen Wolfram (1988): *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Learning from Tactic Steps in Formal Proofs

Thibault Gauthier

Czech Technical University in Prague, Czech Republic

`email@thibaultgauthier.fr`

TacticToe is a machine-learning guided tactical prover which works as follows. By mining a proof assistant library and modifying its proof scripts, tactic invocations are recorded. Abstraction and orthogonalization techniques improve the quality of the created database. A predictor trained on these examples guides TacticToe proof attempts by biasing its searches towards the most promising tactics for each situation. The efficiency of TacticToe is demonstrated on the HOL4 standard library and part of CakeML. By construction, proof scripts generated by TacticToe can be easily inspected and analyzed.

Verifying Bit-vector Invertibility Conditions in Coq – Extended Abstract*

Burak Ekici

University of Innsbruck
Innsbruck, Austria

burak.ekici@uibk.ac.at

Arjun Viswanathan

University of Iowa
Iowa City, USA

arjun-viswanathan@uiowa.edu

Yoni Zohar

Stanford University
Stanford, USA

yoniz@cs.stanford.edu

Clark Barrett

Stanford University
Stanford, USA

barrett@cs.stanford.edu

Cesare Tinelli

University of Iowa
Iowa City, USA

cesare-tinelli@uiowa.edu

This work is a part of an ongoing effort to prove the correctness of invertibility conditions for the theory of fixed-width bit-vectors, which are used to solve quantified bit-vector formulas in the Satisfiability Modulo Theories (SMT) solver CVC4. While many of these were proved in a completely automatic fashion for any bit-width, some were only proved for bit-widths up to 65, even though they are being used to solve formulas over arbitrary bit-widths. In this paper we describe our initial efforts in proving a subset of these invertibility conditions in the Coq proof assistant. We describe the Coq library that we use, as well as the extensions that we introduced to it.

1 Introduction

Reasoning logically about bit-vectors is useful for many applications in hardware and software verification. While Satisfiability Modulo Theories (SMT) solvers are able to reason about bit-vectors of fixed width, they currently require all widths to be expressed concretely (by a numeral) in their input formulas. For this reason, they cannot be used to prove properties of bit-vector operators that are parametric in the bit-width such as, for instance, the associativity of bit-vector concatenation. Proof assistants such as Coq [13], that have direct support for dependent types are better suited for such tasks.

Bit-vector formulas that are parametric in the bit-width arise in the verification of parametric Boolean functions and circuits (see, e.g., [8]). In our case, we are mainly interested in parametric lemmas that are relevant to internal techniques of SMT solvers for the theory of fixed-width bit-vectors. Such techniques are developed a priori for every possible bit-width, even though they are applied on a particular bit-width. Meta-reasoning about the correctness of such solvers then requires bit-width independent reasoning.

An example of the latter kind, which is the focus of the current paper, is the notion of *invertibility conditions* [9] as a basis for a quantifier-instantiation technique to reason about the satisfiability of quantified bit-vector formulas. For a trivial case of an invertibility condition consider the equation $x + s = t$ where x , s and t are variables of the same bit-vector sort, and $+$ is bit-vector addition. In the terminology of Niemetz et al. [9], this equation is “invertible” for x , i.e., solvable for x , for any value of s and t . A general solution is represented by the term $t - s$. Since the solution is unconditional, the invertibility condition for $x + s = t$ is simply the universally true formula \top . The formula stating this fact, referred to here as an *invertibility equivalence*, is $\top \Leftrightarrow \exists x. x + s = t$, a valid formula in the theory of fixed-width

*This work has been partially supported by the Austrian Science Fund (FWF) grant P26201, the European Research Council (ERC) Grant No. 714034 SMART, DARPA award N66001-18-C-4012, and ONR contract N68335-17-C-0558.

bit-vectors for any bit-width n for x , s and t . In contrast, the equation $x \cdot s = t$ is not always invertible for x (\cdot stands for bit-vector multiplication). A necessary and sufficient condition for invertibility is $(-s \mid s) \& t = t$ meaning that the invertibility equivalence $(-s \mid s) \& t = t \Leftrightarrow \exists x. x \cdot s = t$ is valid for any bit-width n for x , s and t [9]. Notice that this invertibility condition involves the operations $\&$, \mid and $-$, and not \cdot that occurs in the literal itself. Niemetz et al. [9] provide a total of 160 invertibility conditions covering several bit-vector operators for both equations and inequations. However, they were able to verify, using SMT solvers, the corresponding invertibility equivalences only for concrete bit-widths up to 65, given the reasoning limitations of SMT solvers mentioned earlier. A recent paper by Niemetz et al. [10] addresses this challenge by translating these invertibility equivalences into quantified formulas over the combined theory of non-linear integer arithmetic and uninterpreted functions — a theory supported by a number of SMT solvers. While partially successful, this approach failed to verify over a quarter of the invertibility equivalences.

In this work, we approach the task of verifying the invertibility equivalences proposed in [9] by proving them interactively with the Coq proof assistant. We extend a rich Coq library for bit-vectors we developed in previous work [6] with additional operators and lemmas to facilitate the task of verifying invertibility equivalences for arbitrary bit-widths, and prove a representative subset of them. Our results offer evidence that proof assistants can support automated theorem provers in meta-verification tasks.

Our Coq library models the theory of fixed-width bit-vectors adopted by the SMT-LIB 2 standard [1].¹ It represents bit-vectors as lists of Booleans. The bit-vector type is dependent on a positive integer that represents the length of the list. Underneath the dependent representation is a simply-typed or *raw* bit-vector type with a size function which is used to explicitly state facts on the length of the list. A functor translates an instance of a raw bit-vector along with specific information about its size into a dependently-typed bit-vector. For this work, we extended the library with the arithmetic right shift operation and the unsigned weak less-than and greater-than predicates and proved 18 invertibility equivalences. We initially proved these equivalences over raw bit-vectors and then used these proofs when proving the invertibility equivalences over dependent bit-vectors, as we explain in Section 4.

The remainder of this paper is organized as follows. After some technical preliminaries in Section 2, we provide an overview of invertibility conditions for the theory of fixed-width bit-vectors in Section 3 and discuss previous attempts to verify them. Then, in Section 4, we describe the bit-vector Coq library and our current extensions to it. In Section 5, we outline how we used the extended library to prove the correctness of a representative subset of invertibility equivalences. We conclude in Section 6 with directions for future work.

2 Preliminaries

We assume the usual terminology of many-sorted first-order logic with equality (see, e.g., [7] for more details). We denote equality by $=$, and use $x \neq y$ as an abbreviation for $\neg(x = y)$. The signature Σ_{BV} of the SMT-LIB 2 theory of fixed-width bit-vectors includes a unique sort for each positive integer n , which we denote here by $\sigma_{[n]}$. For every positive integer n and a bit-vector of width n , the signature includes a constant of sort $\sigma_{[n]}$ in Σ_{BV} representing that bit-vector, which we denote as a binary string of length n . The function and predicate symbols of Σ_{BV} are as described in the SMT-LIB 2 standard. Formulas of Σ_{BV} are built from variables (sorted by the sorts $\sigma_{[n]}$), bit-vector constants, and the function and predicate symbols of Σ_{BV} , along with the usual logical connectives and quantifiers. We write $\psi[x_1, \dots, x_n]$ to represent a formula whose free variables are from the set $\{x_1, \dots, x_n\}$.

¹ The SMT-LIB 2 theory is defined at <http://www.smt-lib.org/theories.shtml>.

The semantics of Σ_{BV} -formulas is given by interpretations that extend a single many-sorted first-order structure so that the domain of every sort $\sigma_{[n]}$ is the set of bit-vectors of bit-width n , and the function and predicate symbols are interpreted as specified by the SMT-LIB 2 standard. A Σ_{BV} -formula is *valid* in the theory of fixed-width bit-vectors if it evaluates to true in every such interpretation.

In what follows, we denote by Σ_0 the sub-signature of Σ_{BV} containing the predicate symbols $<_u, >_u, \leq_u, \geq_u$ (corresponding to strong and weak unsigned comparisons between bit-vectors, respectively), as well as the function symbols $+$ (bit-vector addition), $\&, |, \sim$ (bit-wise conjunction, disjunction and negation), $-$ (2's complement unary negation), and \ll, \gg and \gg_a (left shift, and logical and arithmetical right shifts). We also denote by Σ_1 the extension of Σ_0 with the predicate symbols $<_s, >_s, \leq_s, \geq_s$ (corresponding to strong and weak signed comparisons between bit-vectors, respectively), as well as the function symbols $-, \cdot, \div, \text{mod}$ (corresponding to subtraction, multiplication, division and remainder), and \circ (concatenation). We use 0 to represent the bit-vectors composed of all 0-bits. Its numerical or bit-vector interpretation should be clear from context. Using bit-wise negation \sim , we can express the bit-vectors composed of all 1-bits by ~ 0 .

3 Invertibility Conditions And Their Verification

Many applications rely on bit-precise reasoning and thus can be modeled using the SMT-LIB 2 theory of fixed-width bit-vectors. For certain applications, such as verification of safety properties for programs, quantifier-free reasoning is not enough, and the combination of bit-precise reasoning with the ability to handle quantifiers is needed. Niemetz et al. present a technique to solve quantified bit-vector formulas, which is based on *invertibility conditions* [9]. An invertibility condition for a variable x in a Σ_{BV} -literal $\ell[x, s, t]$ is a formula $IC[s, t]$ such that $\forall s. \forall t. IC[s, t] \Leftrightarrow \exists x. \ell[x, s, t]$ is valid in the theory of fixed-width bit-vectors. For example, consider the bit-vector literal $x \& s = t$ where x, s and t are distinct variables of the same sort. The invertibility condition for x given in [9] is $t \& s = t$.

Niemetz et al. [9] define invertibility conditions for a representative set of literals ℓ having a single occurrence of x , that involve the bit-vector operators of Σ_1 . The soundness of the technique proposed in that work relies on the correctness of the invertibility conditions. Every literal $\ell[x, s, t]$ and its corresponding invertibility condition $IC[s, t]$ induce the *invertibility equivalence*

$$IC[s, t] \Leftrightarrow \exists x. \ell[x, s, t] \tag{1}$$

The correctness of invertibility equivalences should be verified for all possible sorts for the variables x, s, t for which the condition is well sorted. More concretely, for the case where x, s, t are all of sort $\sigma_{[n]}$, say, this means that one needs to prove, for *all* $n > 0$, the validity of

$$\forall s : \sigma_{[n]}. \forall t : \sigma_{[n]}. IC[s, t] \Leftrightarrow \exists x : \sigma_{[n]}. \ell[x, s, t] .$$

This was done in Niemetz et al. [9] using an SMT solver but only for concrete values of n from 1 to 65. A proof of Equation (1) that is parametric in the bit-width n cannot be done with SMT solvers, since they currently only support the theory of *fixed-width* bit-vectors, where Equation (1) cannot even be expressed. To overcome this limitation, a later paper by Niemetz et al. [10] suggested a translation from bit-vector formulas with *parametric* bit-widths to the theory of (non-linear) integer arithmetic with uninterpreted functions. Thanks to this translation, the authors were able to verify, with the aid of SMT solvers for the theory of integer arithmetic with uninterpreted functions, the correctness of 110 out of 160 invertibility equivalences. None of the solvers used in that work were able to prove the remaining equivalences. For

those, it then seems appropriate to use a proof-assistant, as this allows for more intervention by the user who can provide crucial intermediate steps. It goes without saying that even for the 110 invertibility equivalences that were proved, the level of confidence achieved by proving them in a proof-assistant such as Coq would be greater than a verification (without a verified formal proof) by an SMT solver.

In the rest of this paper we describe our initial efforts and future plans for proving the invertibility equivalences, starting with those that were not proved in [10].

4 The Coq Bit-vector Library

In this section, we describe the Coq library we use and the extensions we developed with the goal of formalizing and proving invertibility equivalences. The original library was developed for SMTCoq [6], a Coq plugin that enables Coq to dispatch proofs to external proof-producing solvers. It is used to represent SMT-LIB 2 bit-vectors in Coq. Coq’s own library of bit-vectors [5] was an alternative, but it has only definitions and no lemmas. A more suitable substitute could have been the Bedrock Bit Vectors Library [3] or the SSRBit Library [2]. We chose the SMTCoq library mainly because it was explicitly developed to represent SMT-LIB 2 bit-vectors in Coq and comes with a rich set of lemmas relevant to proving the invertibility equivalences.

The SMTCoq library contains both a simply-typed and dependently-typed theory of bit-vectors implemented as module types. The former, which we also refer to as a theory of *raw bit-vectors*, formalizes bit-vectors as Boolean lists while the latter defines a bit-vector as a Coq record, with its size as the parameter, made of two fields: a Boolean list and a coherence condition to ensure that the parameterized size is indeed the length of the given list. The library also implements a functor module from the simply-typed module to the dependently-typed module establishing a correspondence between the two theories. This way, one can first prove a bit-vector property in the context of the simply-typed theory and then map it to its corresponding dependently-typed one via the functor module. Note that while it is possible to define bit-vectors natively as a dependently-typed theory in Coq and prove their properties there, it would be cumbersome and unduly complex to do dependent pattern matching or case analysis over bit-vector instances because of the complications brought by unification in Coq (which is inherently undecidable). One can try to handle such complications as illustrated by Sozeau [12]. However, we found the two-theory approach of Ekici et al. [6] more convenient in practice for our purposes.

The library adopts the little-endian notation for bit-vectors, thus following the internal representation of bit-vectors in SMT solvers such as CVC4. This makes arithmetic operations easier to perform since the least significant bit of a bit-vector is the head of the list representing it in the *raw* theory.

Out of the 11 bit-vector operators and 10 predicates contained in Σ_1 , the library had support for 8 operators and 6 predicates. The supported predicates, however, can be used to express the other 4. The predicate and function symbols that were not directly supported by the library were the weak inequalities \leq_u , \geq_u , \leq_s , \geq_s and the operators \gg_a , \div , and mod . We extended the library with the operator \gg_a and the predicates \leq_u and \geq_u and redefined \ll and \gg , as explained in Section 5.

We focused on invertibility conditions for literals of the form $x \diamond s \boxtimes t$ and $s \diamond x \boxtimes t$, where x , s and t are variables and \diamond and \boxtimes are respectively function and predicate symbols in $\Sigma_0 \cup \{=, \neq\}$ (invertibility conditions for such literals were found in [9] for the extended signature Σ_1). Σ_0 was chosen as a representative set because it seemed both expressive enough and feasible for proofs in Coq. Such literals, as well as their invertibility conditions, include only operators that are supported by the library (after its extension with \gg_a , \leq_u , and \geq_u).

To demonstrate the intuition and various aspects of the extension of the library, we briefly describe

```

1  Fixpoint ule_list_big_endian (x y : list bool) :=
2    match x, y with
3    | nil, nil => true
4    | nil, _ => false
5    | _, nil => false
6    | xi :: x', yi :: y' => ((eqb xi yi) &&& (ule_list_big_endian x' y'))
7                          || ((negb xi) &&& yi)
8    end.
9
10 Definition ule_list (x y: list bool) :=
11   (ule_list_big_endian (rev x) (rev y)).
12
13 Definition bv_ule (a b : bitvector) :=
14   if @size a =? @size b then
15     ule_list a b
16   else
17     false.

```

Figure 1: Definitions of \leq_u in Coq.

the addition of \leq_u (the definition of \geq_u is similar). The relevant Coq definitions are provided in Figure 1.² Like most other operators, \leq_u is defined in several *layers*. The function `bv_ule`, at the highest layer, ensures that comparisons are between bit-vectors of the same size and then calls `ule_list`. Since we want to compare bit-vectors starting from their most significant bits and the input lists start instead with the least significant bits (because of the little-endian encoding), `ule_list` first reverses the two lists. Then it calls `ule_list_big_endian`, which we consider to be at the lowest layer of the definition. `ule_list_big_endian` then does a lexicographical comparison of the two lists, starting from the most significant bits.

To see why the addition of \leq_u to the library is useful, consider, for example, the following parametric lemma, stating that ~ 0 is the largest unsigned bit-vector of its type:

$$\forall x : \sigma_{[n]}. x \leq_u \sim 0 \quad (2)$$

When not using this explicit operator, we usually rewrite it as:

$$\forall x : \sigma_{[n]}. x <_u \sim 0 \vee x = \sim 0 \quad (3)$$

In such cases, since the definitions of $<_u$ and $=$ have a similar structure to the one in Figure 1, we strip down the layers of $<_u$ and $=$ separately, whereas using \leq_u , we only do this once. Depending on the specific proof at hand, using \leq_u is sometimes more convenient for this reason.

5 Proving Invertibility Equivalences in Coq

In this section we provide specific details about proving invertibility equivalences in Coq. In addition to the bit-vector library described in Section 4, in several proofs of invertibility equivalences we benefited

²Both the library and the proofs of invertibility equivalences can be found at <https://github.com/ekiciburak/bitvector/tree/pxtp2019>. It compiles with coqc-8.9.0.

```

1  Theorem bvashr_ult2_rtl : forall (n : N), forall (s t : bitvector n),
2  (exists (x : bitvector n), (bv_ult (bv_ashr_a s x) t = true)) ->
3  (((bv_ult s t = true) \\/ (bv_slt s (zeros n)) = false) /\
4  (bv_eq t (zeros n)) = false).
5  Proof. intros n s t H.
6      destruct H as ((x, Hx), H).
7      destruct s as (s, Hs).
8      destruct t as (t, Ht).
9      unfold bv_ult, bv_slt, bv_ashr_a, bv_eq, bv in *. cbn in *.
10     specialize (InvCond.bvashr_ult2_rtl n s t Hs Ht); intro STIC.
11     rewrite Hs, Ht in STIC. apply STIC.
12     now exists x.
13  Qed.

```

Figure 2: A proof of one direction of the invertibility equivalence for \gg_a and \ll_u using dependent types.

from CoqHammer [4], a plug-in that aims at extending the automation in Coq by combining machine learning and automated reasoning techniques in a similar fashion to what is done in Isabelle/HOL [11]. Note that one does not need to install CoqHammer in order to build the bit-vector library, since all the proof reconstruction tactics of CoqHammer are included in it.

The natural representation of bit-vectors in Coq is the dependently-typed representation, and therefore the invertibility equivalences are formulated using this representation. As discussed in Section 4, however, proofs in this representation are composed of proofs over simply-typed bit-vectors, which are easier to reason about. Some conversions between the different representations are then needed to lift a proof over raw bit-vectors to one over dependently-typed bit-vectors.

For example, Figure 2 includes a proof of the following direction of the invertibility equivalence for \gg_a and \ll_u :

$$\forall s : \sigma_{[n]}. \forall t : \sigma_{[n]}. (\exists x : \sigma_{[n]}. s \gg_a x \ll_u t) \Rightarrow ((s \ll_u t \vee \neg(s \ll_s 0)) \wedge t \neq 0) \quad (4)$$

In the proof, lines 6–9 transform the dependent bit-vectors from the goal and the hypotheses into simply-typed bit-vectors. Then, lines 10–12 invoke the corresponding lemma for simply-typed bit-vectors (called `InvCond.bvashr_ult2_rtl`) along with some simplifications.

Most of the effort in this project went into proving equivalences over raw bit-vectors. As an illustration, consider the following equivalence over \lll and \ggg_u :

$$\forall s : \sigma_{[n]}. \forall t : \sigma_{[n]}. (t \lll_u \sim 0 \lll s) \Leftrightarrow (\exists x : \sigma_{[n]}. x \lll s \ggg_u t) \quad (5)$$

The left-to-right implication is easy to prove using ~ 0 itself as the witness of the existential proof goal and considering the symmetry between \ggg_u and \lll_u . The proof of the right-to-left implication relies on the following lemma:

$$\forall x : \sigma_{[n]}. \forall s : \sigma_{[n]}. (x \lll s) \leq_u (\sim 0 \lll s) \quad (6)$$

From the right side of the equivalence in Equation (5), we get some x for which $x \lll s \ggg_u t$ holds. Flipping the inequality, we have that $t \lll_u x \lll s$; using this, and transitivity over \lll_u and \leq_u , Lemma 6 gives us the left side of the equivalence in Equation (5).

As mentioned in Section 4, we have redefined the shift operators \lll and \ggg in the library. This was instrumental, for example, in the proof of Equation (6). Figure 3 includes both the original and new

```

1  Definition shl_one_bit (a: list bool) :=
2    match a with
3      | [] => []
4      | _ => false :: removelast a
5    end.
6
7  Fixpoint shl_n_bits (a: list bool) (n: nat) :=
8    match n with
9      | 0 => a
10     | S n' => shl_n_bits (shl_one_bit a) n'
11   end.
12
13  Definition shl_n_bits_a (a: list bool) (n: nat) :=
14    if (n <? length a)%nat then
15      mk_list_false n ++ firstn (length a - n) a
16    else
17      mk_list_false (length a).
18
19  Theorem bv_shl_eq: forall (a b : bitvector), bv_shl a b = bv_shl_a a b.

```

Figure 3: Various definitions of \ll .

definitions of \ll . The definitions of \gg are similar. Originally, \ll was defined using the `shl_one_bit` and the `shl_n_bits` functions. `shl_one_bit` shifts the bit-vector to the left by one bit and is repeatedly called by `shl_n_bits` to complete the shift. The new definition `shl_n_bits_a` uses `mk_list_false` which constructs the necessary list of 0s and appends (`++` in Coq) it to the beginning of the list (because of the little-endian encoding); the bits to be shifted from the original bit-vector are retrieved using the `firstn` function, which is defined in the Coq library for lists. The `nat` type used in Figure 3 is the Coq representation of Peano natural numbers that has 0 and S as its two constructors — as depicted in the pattern match in lines 9 and 10. The theorem at the bottom of Figure 3 allows us to switch between the two definitions when needed. Function `bv_shl` defines the left shift operation using `shl_n_bits` whereas `bv_shl_a` does it using `shl_n_bits_a`.

The new definition uses `firstn` and `++`, over which many necessary properties are already proven in the standard library. This benefits us in manual proofs, and in calls to CoqHammer, since the latter is able to use lemmas from the imported libraries to prove the goals that are given to it. Using this representation, proving Equation (6) reduces to proving Lemmas `bv_ule_1_firstn` and `bv_ule_pre_append`, shown in Figure 4. The proof of `bv_ule_pre_append` benefited from the property `app_comm_cons` from the standard list library of Coq, while `firstn_length_le` was useful in reducing the goal of `bv_ule_1_firstn` to Coq’s equivalent of Equation (2). The statements of the properties mentioned from the standard library are also shown in Figure 4. `mk_list_true` creates a bit-vector that represents ~ 0 , of the length given to it as input, and `bv_ule` is the representation of \leq_u in the bit-vector library. `bv_ule` has output type `bool` (and so we equate terms in which it occurs to `true`), while the functions from the standard library have output type `Prop`. We also have two definitions for \gg_a , and a proof of their equivalence (as done for the other shift operators).

Table 1 summarizes the results of proving invertibility equivalences for invertibility conditions in the signature Σ_0 . In the table, \checkmark means that the invertibility equivalence was successfully verified in Coq but not in [10], while \checkmark means the opposite; \checkmark means that the invertibility equivalence was verified using

```

1 Lemma bv_ule_1_firstn : forall (n : nat) (x : bitvector),
2   (n < length x)%nat ->
3   bv_ule (firstn n x) (firstn n (mk_list_true (length x))) = true.
4
5 Lemma bv_ule_pre_append : forall (x y z : bitvector), bv_ule x y = true ->
6   bv_ule (z ++ x) (z ++ y) = true.
7
8 Theorem app_comm_cons : forall (x y : list A) (a : A), a :: (x ++ y) = (a :: x) ++ y.
9
10 Lemma firstn_length_le : forall l : list A, forall n : nat,
11   n <= length l -> length (firstn n l) = n.

```

Figure 4: Examples of lemmas used in proofs of invertibility equivalences.

$\ell[x]$	$=$	\neq	$<_u$	$>_u$	\leq_u	\geq_u
$\neg x \bowtie t$	✓	✓	✓	✓	✓	✓
$\sim x \bowtie t$	✓	✓	✓	✓	✓	✓
$x \& s \bowtie t$	✓	✓	✓	✓	✓	✓
$x s \bowtie t$	✓	✓	✓	✓	✓	✓
$x \ll s \bowtie t$	✓	✓	✓	✓	✓	✓
$s \ll x \bowtie t$	✓	✓	✓	✓	✓	✓
$x \gg s \bowtie t$	✓	✓	✓	✗	✓	✓
$s \gg x \bowtie t$	✓	✓	✓	✓	✓	✓
$x \gg_a s \bowtie t$	✓	✓	✓	✓	✓	✓
$s \gg_a x \bowtie t$	✓	✓	✓	✓	✓	✓
$x + s \bowtie t$	✓	✓	✓	✓	✓	✓

Table 1: Proved invertibility equivalences in Σ_0 where \bowtie ranges over the given predicate symbols.

both approaches, and \times means that it was verified with neither. We successfully proved all invertibility equivalences over $=$ that are expressible in Σ_0 , including 4 that were not proved in [10]. For the rest of the predicates, we focused only on the 8 invertibility equivalences that were not proved in [10], and succeeded in proving 7 of them. Overall, these results strictly improve the results of [10], as we were able to prove 11 additional invertibility equivalences in Coq. Taking into account our work together with [10], only one invertibility equivalence for the restricted signature is not fully proved yet, the one for the literal $x \gg s >_u t$, although one direction of the equivalence, namely $IC[s, t] \Rightarrow \exists x. \ell[x, s, t]$, was successfully proved both in Coq and in [10].

6 Conclusion and Future Work

We have described our work-in-progress on verifying bit-vector invertibility conditions in the Coq proof assistant, which required extending a bit-vector library in Coq. The most immediate direction for future

work is proving more of the invertibility equivalences supported by the bit-vector library. In addition, we plan to extend the library so that it supports the full syntax in which invertibility conditions are expressed, namely Σ_1 . We expect this to be useful also for verifying properties about bit-vectors in other applications.

References

- [1] Clark Barrett, Aaron Stump & Cesare Tinelli (2010): *The SMT-LIB Standard: Version 2.0*. In A. Gupta & D. Kroening, editors: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*.
- [2] Arthur Blot, Pierre-Evariste Dagand, & Julia Lawall: *Bit Sequences and Bit Sets Library*. Available at <https://github.com/pedagand/ssrbit>.
- [3] Tej Chajed, Haogang Chen, Adam Chlipala, Joonwon Choi, Andres Erbsen, Jason Gross, Samuel Gruetter, Frans Kaashoek, Alex Konradi, Gregory Malecha, Duckki Oe, Murali Vijayaraghavan, Nickolai Zeldovich & Daniel Ziegler: *Bedrock Bit Vectors Library*. Available at <https://github.com/mit-plv/bbv>.
- [4] Lukasz Czajka & Cezary Kaliszyk (2018): *Hammer for Coq: Automation for Dependent Type Theory*. *J. Autom. Reasoning* 61(1-4), pp. 423–453, doi:[10.1007/s10817-018-9458-4](https://doi.org/10.1007/s10817-018-9458-4).
- [5] Jean Duprat: *Library Coq.Bool.Bvector*. Available at <https://coq.inria.fr/library/Coq.Bool.Bvector.html>.
- [6] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds & Clark Barrett (2017): *SMTCoq: A Plug-In for Integrating SMT Solvers into Coq*. In: *Proceedings of 29th International Conference on Computer Aided Verification (CAV 2017)*, *Lecture Notes in Computer Science* 10427, Springer, pp. 126–133, doi:[10.1007/s10703-012-0163-3](https://doi.org/10.1007/s10703-012-0163-3)
- [7] Herbert B. Enderton (2001): *Chapter TWO - First-Order Logic*. In Herbert B. Enderton, editor: *A Mathematical Introduction to Logic (Second Edition)*, second edition edition, Academic Press, Boston, pp. 67 – 181, doi:[10.1016/B978-0-08-049646-7.50008-4](https://doi.org/10.1016/B978-0-08-049646-7.50008-4).
- [8] Aarti Gupta & Allan L. Fisher (1993): *Representation and Symbolic Manipulation of Linearly Inductive Boolean Functions*. In: *Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design, ICCAD '93*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 192–199. Available at <http://dl.acm.org.stanford.idm.oclc.org/citation.cfm?id=259794.259827>.
- [9] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett & Cesare Tinelli (2018): *Solving Quantified Bit-Vectors Using Invertibility Conditions*. In: *Proceedings of 30th International Conference on Computer Aided Verification (CAV 2018)*, pp. 236–255, doi:[10.1007/978-3-319-96142-2_16](https://doi.org/10.1007/978-3-319-96142-2_16).
- [10] Aina Niemetz, Mathias Preiner, Andrew Reynolds Yoni Zohar, Clark Barrett & Cesare Tinelli (2019): *Towards Bit-Width-Independent Proofs in SMT Solvers*. To appear in the proceedings of CADE-27.
- [11] Tobias Nipkow, Lawrence C Paulson & Markus Wenzel (2002): *Isabelle/HOL: a proof assistant for higher-order logic*. *Lecture Notes in Computer Science* 2283, Springer Science & Business Media, doi:[10.1007/3-540-45949-9_6](https://doi.org/10.1007/3-540-45949-9_6)
- [12] Matthieu Sozeau (2010): *Equations: A Dependent Pattern-Matching Compiler*. In: *Proceedings of the 1st International Conference on Interactive Theorem Proving (ITP 2010)*, pp. 419–434, doi:[10.1007/978-3-642-14052-5_29](https://doi.org/10.1007/978-3-642-14052-5_29).
- [13] The Coq development team (2019): *The Coq Proof Assistant Reference Manual Version 8.9*. Available at <https://coq.inria.fr/distrib/current/refman/>.

EKSTRAKTO

A tool to reconstruct *Dedukti* proofs from TSTP files (extended abstract)

Mohamed Yacine El Haddad
CNRS

Guillaume Burel
ENSIE

Frédéric Blanqui
INRIA

LSV, CNRS, ENS Paris-Saclay, Université Paris-Saclay

Proof assistants often call automated theorem provers to prove subgoals. However, each prover has its own proof calculus and the proof traces that it produces often lack many details to build a complete proof. Hence these traces are hard to check and reuse in proof assistants. DEDUKTI is a proof checker whose proofs can be translated to various proof assistants: Coq, HOL, Lean, Matita, PVS. We implemented a tool that extracts TPTP subproblems from a TSTP file and reconstructs complete proofs in DEDUKTI using automated provers able to generate DEDUKTI proofs like ZenonModulo or ArchSAT. This tool is generic: it assumes nothing about the proof calculus of the prover producing the trace, and it can use different provers to produce the DEDUKTI proof. We applied our tool on traces produced by automated theorem provers on the CNF problems of the TPTP library and we were able to reconstruct a proof for a large proportion of them, significantly increasing the number of DEDUKTI proofs that could be obtained for those problems.

1 Introduction

In order to discharge more burden from the users of interactive theorem provers, and thus to widen the use of these tools, it is crucial to automate them more. To achieve this goal, in the process of checking the validity of formulas, proof assistants could use an external theorem prover to automate their tasks and obtain a proof of a specific formula. Once a proof is found, the proof assistant applies this proof on the current goal and tells the user that all is done in background. However, this can work only if the prover builds a complete proof that is easily checkable by the proof assistant. We distinguish two families of automated theorem provers: some provers, like *ZenonModulo* [5] and *ArchSAT* [3], output complete proofs but are not very efficient to find a proof; others, like *E prover* [7] and *ZipperPosition* [4], are more powerful but return only proof traces, i.e. proofs with less details.

In this paper we are interested in first-order automated theorem provers which can return TSTP [8] traces. We will use DEDUKTI [1] as proof checker because DEDUKTI files can be translated to many other proof assistants (Coq, HOL, Lean Matita, PVS) [10].

We start by presenting the TPTP/TSTP formats with an example. Then, we describe how proofs and formulas are encoded in DEDUKTI. We then present our solution implemented in a tool named EKSTRAKTO in two steps: extraction of sub-problems and proof reconstruction. Finally, we conclude and give some perspectives.

2 TPTP

TPTP [8] is a standard library of problems to test automated theorem provers [9]. Each TPTP file represents a problem in propositional, first-order or higher-order logic. We distinguish the type of formulas

by using one of the keywords: CNF, FOF, TFF and THF, corresponding respectively to mono-sorted first-order formulas in clausal normal form, general mono-sorted first-order formulas, typed first-order formulas, and typed higher-order formulas.

In this work, we restrict our attention to CNF formulas since their proofs use logical consequences only, which is not the case of FOF formulas (e.g. Skolemisation).

Apart from an include instruction, each line in a TPTP file is a declaration of a formula given with its role, e.g. axiom, hypothesis, definition or conjecture:

```
cnf(name, role, formula, information).
```

TSTP [8] is a library of solutions to TPTP problems. In this paper, we call a TSTP file a trace. It is obtained after running an automated theorem prover on a TPTP problem. The syntax used in a TSTP file is the same as TPTP except for the content of the *information* field. This field contains general information about how the current formula is obtained. Here is the grammar used to describe a source in the *information* field:

```
<source>           ::= <dag_source> | [ <sources> ] | ...
<dag_source>       ::= <name> | inference(..., ..., <inference_parents>)
<inference_parents> ::= [] | [ <sources> ]
<sources>          ::= <source> (, <source>)*
```

For our purpose only 3 cases are of interest as shown in the grammar above:

- 1) When it is the name of a formula previously declared.
- 2) When it is a list of several sources:

```
[s_0, s_1, ..., s_n]
```

- 3) When it is an inference:

```
inference(name, infos, [s_0, s_1, ..., s_n])
```

The name of the inference refers to the name of the rule used by the prover to prove the current step. The *infos* field contains more information about the inference like status, inference name, etc. Note that each s_i is a source and therefore can contain sub-inferences.

Here is an example of a TSTP file obtained after running *E prover* on the TPTP problem SET001-1:

SET001-1.p

```
cnf(c_0, axiom,
    ( subset(X1,X2)
      | ~ equal_sets(X1,X2) ) ).
cnf(c_1, hypothesis,
    ( equal_sets(b,bb) ) ).
cnf(c_2, axiom,
    ( member(X1,X3)
      | ~ member(X1,X2)
      | ~ subset(X2,X3) ) ).
cnf(c_3, negated_conjecture,
    ( ~ member(element_of_b,bb) ) ).
cnf(c_4, hypothesis,
```

```

      ( member(element_of_b,b) )) .
cnf(c_5,hypothesis ,
    ( subset(b,bb) ),
      inference(spm,[status(thm)],[c_0,c_1])) .
cnf(c_6,hypothesis ,
    ( member(X1,bb)
      | ~ member(X1,b) ),
      inference(spm,[status(thm)],[c_2,c_5])) .
cnf(c_7,negated_conjecture ,
    ( $false ),
      inference(cn,[status(thm)],[inference(rw,[status(thm)],[
      inference(spm,[status(thm)],[c_3,c_6]),c_4])]),
      [proof])) .

```

We can represent this trace as the following tree:

$$\frac{\frac{\frac{\frac{}{\vdash \text{Form}(c_3)}{\vdash \text{Form}(c_2)} \text{spm}}{\vdash \text{Form}(c_6)} \text{spm}}{\vdash \text{Form}(c_7)} \text{spm}}{\vdash \text{Form}(c_7)} \text{cn}}{\vdash \text{Form}(c_4)} \text{rw}}{\vdash \text{Form}(c_7)} \text{cn}$$

where:

```

Form(c_0) = subset(X1,X2) | ~equal_sets(X1,X2)
Form(c_1) = equal_sets(b,bb)
Form(c_2) = member(X1,X3) | ~member(X1,X2) | ~subset(X2,X3)
Form(c_3) = ~member(element_of_b,bb)
Form(c_4) = member(element_of_b,b)
Form(c_5) = subset(b,bb)
Form(c_6) = member(X1,bb) | ~member(X1,b)
Form(c_7) = $false

```

3 First-order logic in DEDUKTI

DEDUKTI is a proof checker based on the $\lambda\Pi$ -calculus modulo rewriting [1]. In DEDUKTI, one can declare (dependent) types and function symbols, and rewriting rules for defining these symbols. We describe how a formula and its proof are encoded in DEDUKTI using the *Curry-Howard* correspondence, i.e., we interpret formulas as types and their proofs as terms. In the following, we recall the encoding of first-order logic in DEDUKTI, as described in [1]. This encoding is used in *ZenonModulo*, which is an extension to rewriting of the automated theorem prover *Zenon* [2]. *ZenonModulo* outputs DEDUKTI files after having found a proof using the tableaux method. The following file defines the type of sorts, the type of terms, the type of formulas and then the type of proofs.

zen.lp

```

symbol sort : TYPE          // Dedukti type for sorts
symbol  $\iota$  : sort          // default sort

symbol term : sort  $\Rightarrow$  TYPE // Dedukti type for sorted terms

symbol prop : TYPE          // Dedukti type for formulas
symbol  $\perp$  : prop
symbol  $\top$  : prop
symbol  $\neg$  : prop  $\Rightarrow$  prop
symbol  $\wedge$  : prop  $\Rightarrow$  prop  $\Rightarrow$  prop
symbol  $\vee$  : prop  $\Rightarrow$  prop  $\Rightarrow$  prop
symbol  $\Rightarrow$  : prop  $\Rightarrow$  prop  $\Rightarrow$  prop
symbol  $\forall$  :  $\forall$  a, (term a  $\Rightarrow$  prop)  $\Rightarrow$  prop
symbol  $\exists$  :  $\forall$  a, (term a  $\Rightarrow$  prop)  $\Rightarrow$  prop
symbol  $\doteq$  :  $\forall$  a, term a  $\Rightarrow$  term a  $\Rightarrow$  prop

symbol Proof : prop  $\Rightarrow$  TYPE // interprets formulas as types
rule Proof ( $\Rightarrow$  &a &b)  $\rightarrow$  Proof &a  $\Rightarrow$  Proof &b
// rewriting rule defining the type of proofs for  $\Rightarrow$ 

```

Now, for each TSTP file, we generate a Dedukti file defining its signature by declaring a Dedukti symbol f for each function symbol f of the TSTP file:

SET001-1.lp

```

symbol element_of_b : zen.term  $\iota$ 
symbol subset       : zen.term  $\iota$   $\Rightarrow$  zen.term  $\iota$   $\Rightarrow$  zen.prop
symbol b            : zen.term  $\iota$ 
symbol member       : zen.term  $\iota$   $\Rightarrow$  zen.term  $\iota$   $\Rightarrow$  zen.prop
symbol bb           : zen.term  $\iota$ 
symbol equal_sets   : zen.term  $\iota$   $\Rightarrow$  zen.term  $\iota$   $\Rightarrow$  zen.prop

```

Hence, every formula of first-order logic can be represented in DEDUKTI by using the function φ defined as follows:

$$\begin{aligned}
\varphi(x) &:= x \\
\varphi(f(t_1, t_2, \dots, t_n)) &:= f \varphi(t_1) \varphi(t_2) \dots \varphi(t_n) \\
\varphi(\perp) &:= \perp \\
\varphi(\top) &:= \top \\
\varphi(\neg A) &:= \neg \varphi(A) \\
\varphi(A \wedge B) &:= \varphi(A) \wedge \varphi(B) \\
\varphi(A \vee B) &:= \varphi(A) \vee \varphi(B) \\
\varphi(A \Rightarrow B) &:= \varphi(A) \Rightarrow \varphi(B) \\
\varphi(\forall x A) &:= \forall_i (\lambda x, \varphi(A)) \\
\varphi(\exists x A) &:= \exists_i (\lambda x, \varphi(A)) \\
\varphi(x = y) &:= \varphi(x) \doteq_i \varphi(y)
\end{aligned}$$

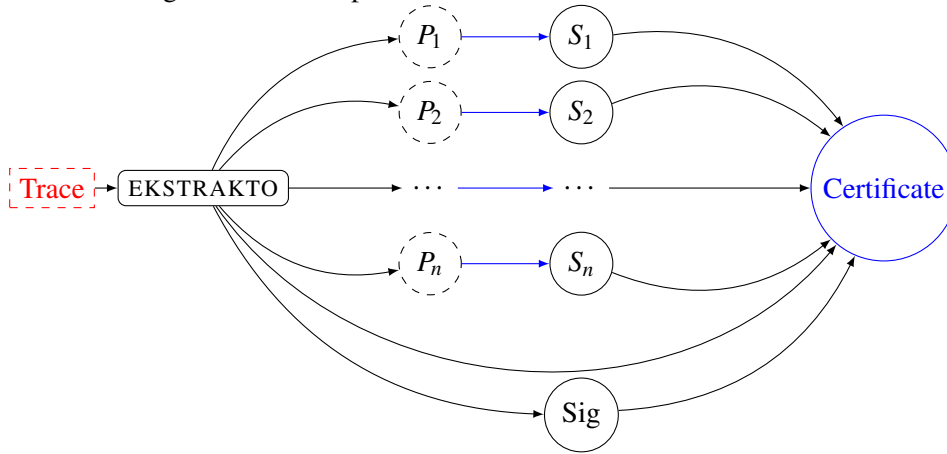
For example,

$$\varphi(\forall X_1, \forall X_2, s(X_1, X_2) \vee \neg e(X_1, X_2)) := \dot{\forall}_t(\lambda X_1, \dot{\forall}_t(\lambda X_2, (s X_1 X_2) \dot{\vee} \dot{\neg}(e X_1 X_2)))$$

For every formula A , its proof in DEDUKTI is a term that has the type $Proof(\varphi(A))$. One can define a similar embedding for proofs as the one we presented for first-order formulas, as shown in [1].

4 Architecture

In this section, we explain in details how EKSTRAKTO works. In order to produce a DEDUKTI proof from a TSTP file, EKSTRAKTO extracts a TPTP problem for each formula declaration containing at least one inference, and calls *ZenonModulo* (or any other automated prover producing DEDUKTI proofs, see discussion below) on each generated problem to get a DEDUKTI proof for this problem. If the external prover succeeds to find a proof of all the generated problems, then we combine those proofs in another DEDUKTI file to get a DEDUKTI proof of the whole TSTP file.



4.1 Extracting TPTP problems

To extract a TPTP problem from a trace step, we need to find the premises used in it. We define the function \mathcal{P} which takes a TSTP source as input and returns the set of premises used by the prover:

$$\mathcal{P}(name) = \{name\}$$

$$\mathcal{P}([s_0, s_1, \dots, s_n]) = \bigcup_{i=0}^n \mathcal{P}(s_i)$$

$$\mathcal{P}(inference(name, infos, [s_0, s_1, \dots, s_n])) = \bigcup_{i=0}^n \mathcal{P}(s_i)$$

Note that if we have an inference t inside another one, say s , we will repeat the process for each sub-inference and omit s from the set of premises, i.e., if we represent an inference step by a proof tree we take only the leaves of this tree as premises.

We omit all information that is not needed (*status, name, ...*). In particular we do not consider the inference name field. Even if it could be used to fine-tune the problem, we prefer to ignore it in order to remain generic since the names are specific to the prover that produced the trace. Hence, we have:

$$\mathcal{P}(\text{inference}([\text{inference}([\text{inference}([c_3, c_6]), c_4])])) = \{c_3, c_6, c_4\}$$

After getting all the premises used for proving $\text{Form}(\text{name})$, say $\text{name}_0, \dots, \text{name}_k$, we generate the following TPTP problem:

$$\text{Form}(\text{name}_0) \Rightarrow \dots \Rightarrow \text{Form}(\text{name}_k) \Rightarrow \text{Form}(\text{name})$$

Note that the generated TPTP problem is a FOF formula. The reason of this choice is to keep the same formula when we combine the sub-proofs. If we generated a CNF problem, then we would need to negate the goal and it would be more complex to reconstruct the proof.

Since we are using FOF formulas in sub-problems that are obtained from a CNF trace, we need to quantify over each free variable to get a closed formula.

In our example, there are 3 steps (colored in blue in the file SET001-1.p above). EKSTRAKTO will generate the following 3 first-order formulas:

$$\text{Form}(c_0) \Rightarrow \text{Form}(c_1) \Rightarrow \text{Form}(c_5)$$

$$\text{Form}(c_2) \Rightarrow \text{Form}(c_5) \Rightarrow \text{Form}(c_6)$$

$$\text{Form}(c_3) \Rightarrow \text{Form}(c_6) \Rightarrow \text{Form}(c_4) \Rightarrow \text{Form}(c_7)$$

Each formula will be written in a separate TPTP file as follows:

c_5.p

```
fof(c_5, conjecture, (
  (![X1, X2] : (s (X1, X2) | ~equal_sets (X1, X2)))
  => ((equal_sets (b, bb))
  => (subset (b, bb)))).
```

c_6.p

```
fof(c_6, conjecture, (
  (![X1, X2, X3] : (member (X1, X3) | ~member (X1, X2)
  | ~subset (X2, X3)))
  => ((subset (b, bb))
  => (![X1] : (member (X1, bb) | ~member (X1, b))))).
```

c_7.p

```
fof(c_7, conjecture, (
  (~member (element_of_b, bb))
  => ((![X1] : (member (X1, bb) | ~member (X1, b)))
  => ((member (element_of_b, b))
  => ($false)))).
```

4.2 Proof reconstruction

If the automated theorem prover succeeds to solve all the generated TPTP problems, then we can reconstruct a proof in DEDUKTI directly by using the proof tree of the trace that we are trying to certify and all the proofs of the sub-problems. The proof term of each sub-problem is irrelevant since it has the right type.

The global proof is reconstructed from each sub-proof. We just need to apply each proof term of a sub-proof to its premises by following the proof tree of the TSTP file. Indeed, the type of the sub-proof of $\text{Form}(\text{name})$ using premises $\text{name}_0, \dots, \text{name}_k$ is

$$\text{zen.Proof } (\Rightarrow \varphi(\text{Form}(\text{name}_0)) (\Rightarrow \varphi(\text{Form}(\text{name}_1)) \dots (\Rightarrow \varphi(\text{Form}(\text{name}_k)) \varphi(\text{Form}(\text{name}))) \dots))$$

Thanks to the rule given in `zen.lp` in Section 3, this type is convertible to

$$\text{zen.Proof } (\varphi(\text{Form}(\text{name}_0))) \Rightarrow \dots \Rightarrow \text{zen.Proof } (\varphi(\text{Form}(\text{name}_k))) \Rightarrow \text{zen.Proof } (\varphi(\text{Form}(\text{name})))$$

Hence, the proof term of a sub-problem is a function whose arguments are proofs of the premises and which returns a proof of its conclusion. Since we are handling only CNF formulas, the proof that we want to reconstruct at the end is always a proof of \perp . Before applying those proof terms we need to declare our hypotheses. With our example file we get:

`proof_SET001-1.lp`

```

definition proof_trace
  (hyp_c_0 : zen.Proof (φ(Form(c_0))))
  (hyp_c_1 : zen.Proof (φ(Form(c_1))))
  (hyp_c_2 : zen.Proof (φ(Form(c_2))))
  (hyp_c_3 : zen.Proof (φ(Form(c_3))))
  (hyp_c_4 : zen.Proof (φ(Form(c_4))))
  : zen.Proof ⊥
  :=
  let lemma_c_5 = c_5.delta hyp_c_0 hyp_c_1 in
  let lemma_c_6 = c_6.delta hyp_c_2 lemma_c_5 in
  let lemma_c_7 = c_7.delta hyp_c_3 lemma_c_6 hyp_c_4 in
  lemma_c_7

```

where *delta* is the name of the proof term in each file.

All this has been implemented in a tool called EKSTRAKTO¹ consisting of 2,000 lines of OCaml.

5 Experiments

We run the *E prover* (version 2.1) on the set of CNF problems of TPTP library v7.2.0 (7922 files) with 2GB of memory space and a timeout of 5 minutes. We obtained 4582 TSTP files. On these TSTP files, EKSTRAKTO generated 362556 TPTP files. *ZenonModulo* generated a DEDUKTI proof for 90% of these files, *ArchSAT* generated 96% and the union of both produced 97% DEDUKTI proofs:

¹<https://github.com/elhaddadyacine/ekstrakto>

Table 1: Percentage of DEDUKTI proofs on the 362556 extracted TPTP files

Prover	% TPTP
<i>ZenonModulo</i>	90%
<i>ArchSAT</i>	96%
$ZenonModulo \cup ArchSAT$	97%

However, as it suffices that no DEDUKTI proof is found for only one TPTP file for getting no global proof, EKSTRAKTO can generate a complete proof for only 48% of TSTP files using *ZenonModulo*, 61% using *ArchSAT* and 72% using at least one of them:

Table 2: Percentage of DEDUKTI proofs on the 4582 TSTP files generated by *E prover*

Prover	% TSTP
<i>ZenonModulo</i>	48%
<i>ArchSAT</i>	61%
$ZenonModulo \cup ArchSAT$	71%

Consequently, we are now able to produce 2189 DEDUKTI proofs from the TPTP library using *E prover* and *Zenon Modulo* (resp. 2793 using *E prover* and *ArchSAT* and 3285 using *E prover*, *Zenon Modulo* and *ArchSAT*), whereas under the same conditions, *Zenon Modulo* alone is only able to produce 1026 DEDUKTI proofs (resp. 500 for *ArchSAT* alone).

Sometimes, *ZenonModulo* and *ArchSAT* fail to find a proof even if the sub-problem is simpler than the main one. This is justified by the fact that the proof calculus used in *ZenonModulo* and *ArchSAT* is based on a different method from the one used in *E prover*. In fact, some steps that are trivial for a prover based on resolution or superposition may not be trivial for *ZenonModulo* or *ArchSAT* which use the tableaux method.

iProverModulo is another candidate to prove TSTP steps, but it performs some transformations before outputting a DEDUKTI proof. Therefore the proof reconstruction is hard in the sense that we need to justify each transformation made by *iProverModulo*.

6 Conclusion and perspectives

We have presented a tool that reconstructs proofs generated by first-order theorem provers. We described how proofs and formulas are represented in DEDUKTI and how we can implement a simple proof reconstruction.

The advantage of EKSTRAKTO is to be generic since it does not depend on the rules used by the automated prover to find the proof. Another advantage is the fact that the proofs are expressed in DEDUKTI, i.e., we can translate them to many other systems (Coq, HOL, Lean, Matita, PVS).

In our experiments, we used *ZenonModulo* and *ArchSAT* to prove each trace step since they are tools that produce DEDUKTI proof terms.

EKSTRAKTO should be extended to handle non-provable steps like Skolemisation. This latter technique could possibly be implemented using the method described in [6]. We should also be more generic, by supporting more features of TSTP like typed formulas and definitions introduced by the prover.

Acknowledgements. The authors thank the anonymous reviewers for their useful comments. This

research was partially supported by the Labex DigiCosme (ANR11LABEX0045DIGICOSME) operated by ANR as part of the program "Investissement d'Avenir" Idex ParisSaclay (ANR11IDEX000302).


References

- [1] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant & Ronan Saillard: *Dedukti: a Logical Framework based on the $\lambda\Pi$ -Calculus Modulo Theory*. Available at <http://www.lsv.fr/~dowek/Publi/expressing.pdf>.
- [2] Richard Bonichon, David Delahaye & Damien Doligez (2007): *Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs*. In: *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, pp. 151–165, doi:10.1007/978-3-540-75560-9_13.
- [3] Guillaume Bury, Simon Cruanes & David Delahaye (2018): *SMT Solving Modulo Tableau and Rewriting Theories*. Available at <https://hal.archives-ouvertes.fr/hal-02083232>.
- [4] Simon Cruanes (2015): *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond*. Theses, École polytechnique. Available at <https://hal.archives-ouvertes.fr/tel-01223502>.
- [5] David Delahaye, Damien Doligez, Frédéric Gilbert, Pierre Halmagrand & Olivier Hermant (2013): *Zenon Modulo: When Achilles Outruns the Tortoise Using Deduction Modulo*. In Ken McMillan, Aart Middeldorp & Andrei Voronkov, editors: *Logic for Programming, Artificial Intelligence, and Reasoning*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 274–290, doi:10.1007/978-3-642-45221-5_20.
- [6] Gilles Dowek & Benjamin Werner: *A constructive proof of Skolem theorem for constructive logic*. Available at <http://www.lsv.fr/~dowek/Publi/skolem.pdf>.
- [7] Stephan Schulz (2013): *System Description: E 1.8*. In Ken McMillan, Aart Middeldorp & Andrei Voronkov, editors: *Proc. of the 19th LPAR, Stellenbosch, LNCS 8312*, Springer, doi:10.1007/978-3-642-45221-5_49.
- [8] G. Sutcliffe (2017): *The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0*. *Journal of Automated Reasoning* 59(4), pp. 483–502, doi:10.1007/s10817-017-9407-7.
- [9] Geoff Sutcliffe (2018): *The 9th IJCAR Automated Theorem Proving System Competition - CASC-J9*. *AI Commun.* 31(6), pp. 495–507, doi:10.3233/AIC-180773.
- [10] François Thiré (2018): *Sharing a Library between Proof Assistants: Reaching out to the HOL Family*. In Frédéric Blanqui & Giselle Reis, editors: *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTTP@FSCD 2018, Oxford, UK, 7th July 2018.*, *EPTCS 274*, pp. 57–71, doi:10.4204/EPTCS.274.5.

Reconstructing veriT Proofs in Isabelle/HOL

Mathias Fleury 

Max-Planck-Institut für Informatik,
Saarland Informatics Campus, Saarbrücken, Germany
mathias.fleury@mpi-inf.mpg.de
Graduate School of Computer Science,
Saarland Informatics Campus, Saarbrücken, Germany
s8mafleu@stud.uni-saarland.de

Hans-Jörg Schurr 

University of Lorraine, CNRS, Inria, and
LORIA, Nancy, France
hans-jorg.schurr@inria.fr

Automated theorem provers are now commonly used within interactive theorem provers to discharge an increasingly large number of proof obligations. To maintain the trustworthiness of a proof, the automatically found proof must be verified inside the proof assistant. We present here a reconstruction procedure in the proof assistant Isabelle/HOL for proofs generated by the satisfiability modulo theories solver veriT which is part of the `smt` tactic. We describe in detail the architecture of our improved reconstruction method and the challenges we faced in designing it. Our experiments show that the veriT-powered `smt` tactic is regularly suggested by Sledgehammer as the fastest method to automatically solve proof goals.

1 Introduction

Proof assistants are used in verification, formal mathematics, and other areas to provide trustworthy and machine-checkable formal proofs of theorems. Proof automation allows the user to focus on the core of their argument by reducing the burden of manual proof. A successful approach to automation is to invoke an external automatic theorem prover (ATP), such as a satisfiability modulo theories (SMT) solver [5] and to reconstruct any generated proofs using the proof assistant’s inference kernel. The usefulness of this approach depends on the encoding of the proof goal into the language of the ATP, the capabilities of the ATP, the quality of the generated proof output, and the reconstruction routine itself.

In the proof assistant Isabelle/HOL this approach is implemented in the `smt` tactic [8].¹ This tactic encodes the proof goal into the SMT-LIB language [4] and calls the SMT solver Z3 [14]. If Z3 is successful in finding a proof of the input problem, the generated proof is reconstructed inside Isabelle. The proof format, and hence the reconstruction process, is specific to Z3. If the reconstruction is successful, the initial proof goal holds in the Isabelle/HOL logic. The reconstruction, however, might fail due to errors (either due to a weakness in the reconstruction or due to errors to the solver) or timing out.

We have previously developed [2, 3] a prototype to reconstruct proofs generated by the SMT solver veriT [9]. In this paper we have extended these works with proper support of term sharing, tested it on a much larger scale, and present the reconstruction method in more detail. Furthermore, we reworked the syntax of the proof output to adhere stronger to the SMT-LIB standard. Given the variety of capabilities between ATPs, a greater diversity in supported systems increases the number of proof goals which can be solved by automated tools. Moreover, the fine-grained proofs produced by veriT might allow for a higher success rate in reconstruction. Lastly, the reconstruction efforts provide valuable insights for the design of proof formats.

¹Technically, `smt` is a proof method, but the difference (whether it requires an Isabelle context) does not matter here.

Similar to the proofs generated by Z3, veriT’s proofs are based on the SMT-LIB language, but are otherwise different. Proofs are a list of indexed steps which can reference steps appearing before them in the list. Steps without references are tautologies and assumptions. The last step is always the deduction of the empty clause. Furthermore, steps can be marked as subproofs, which are used for local assumptions and to reason about bound variables. To shorten the proof length, we use term sharing, which is implemented using the standard SMT-LIB name annotation mechanism. Major differences to the proof format used by Z3 are the fine-grained steps for Skolemization and the presence of steps for the manipulations of bound variables [2].

Our reconstruction routine inside Isabelle is structured as a pipeline. Once the proof is parsed into a datatype and the term sharing is unfolded, the SMT-LIB terms are translated into Isabelle terms. At this point the proof can be replayed step-by-step. Special care has to be taken to handle Skolem terms, subproofs, and the unfolding of the encoding into the first-order logic of the SMT solver.

We validate our reconstruction approach in two ways. First, we replace the calls of the `smt` tactic that are currently using Z3 by veriT. Second, we use Sledgehammer to validate the utility of veriT as a backend solver for the `smt` tactic. Sledgehammer uses external ATPs to find a collection of theorems from the background theory which are sufficient to prove the goal. It then tests a collection of automated tactics on this set of theorems and suggests the fastest successful tactic to the user. On theories from the Archive of Formal Proofs, Sledgehammer suggests the usage of the veriT-powered `smt` tactic on a significant number of proof steps. This suggests that the overall checking speed can be improved by switching to the veriT-powered `smt` tactic at these points.

2 The Proofs Generated by veriT

veriT is a CDCL(T)-based satisfiability modulo theories solver. It uses the SMT-LIB language as input and output language and also utilizes the many-sorted classical first-order logic defined by this language. If requested by the user, veriT outputs a proof if it can deduce that the input problem is unsatisfiable. In proof production mode, veriT supports the theory of uninterpreted functions, the theory of linear integer and real arithmetic, and quantifiers.

We assume the reader is familiar with many-sorted classical first-order logic. To simplify the notation we will omit the sort of terms, except when absolutely needed. The available sorts depend on the selected SMT-LIB theory and can also be extended by the user, but a distinguished **Bool** sort is always available. We use the symbols x, y, z for variables, f, g, h for functions, and P, Q for predicates, i.e., functions with result sort **Bool**. The symbols r, s, t, u stand for terms. The symbols φ, ψ denote formulas, i.e., terms of sort **Bool**. We use σ to denote substitutions and $t\sigma$ to denote the application of the substitution on the term t . To denote the substitution which maps x to t we write $[t/x]$. We use $=$ to denote syntactic equality and \simeq to denote the sorted equality predicate. Since veriT implicitly removes double negations, we also use the notion of complementary literals very liberally: $\varphi = \bar{\psi}$ holds if the terms obtained after removing all leading negations from φ and $\bar{\psi}$ are syntactically equal and the number of leading negations is even for φ and odd for $\bar{\psi}$, or vice versa.

A proof generated by veriT is a list of steps. A step consists of an index $i \in \mathbb{N}$, a formula φ , a rule name R taken from a set of possible rules, a possibly empty set of premises $\{p_1, \dots, p_n\}$ with $p_i \in \mathbb{N}$, a rule-dependent and possibly empty list of arguments $[a_1, \dots, a_m]$, and a context Γ . The arguments a_i are either terms or tuples (x_i, t_i) where x_i is a variable and t_i is a term. The interpretation of the arguments is rule specific. The context is a possibly empty list $[c_1, \dots, c_l]$, where c_i stands for either a variable or a variable-term tuple (x_i, t_i) . A context denotes a substitution as described in section 2.1. Every proof ends

with a step with the empty clause as the step term and empty context. The list of premises only references earlier steps, such that the proof forms a directed acyclic graph. In Appendix A we provide an overview of all proof rules used by veriT.

To mimic the actual proof text generated by veriT we will use the following notation to denote a step:

$$c_1, \dots, c_l \triangleright i. \quad \varphi \quad (\text{RULE}; p_1, \dots, p_n; a_1, \dots, a_m)$$

If an element of the context c_i is of the form (x_i, t_i) , we will write $x_i \mapsto t_i$. If an element of the arguments a_i is of this form we will write $x_i := t_i$. Furthermore, the proofs can utilize Hilbert's choice operator ε . Choice acts like a binder. The term $\varepsilon x. \varphi$ stands for a value v , such that $\varphi[v/x]$ is true if such a value exists. Any value is possible otherwise.

The proof format used by veriT has been discussed in prior publications: the fundamental ideas behind the proof format have been discussed in [6]; proposed rules for quantifier instantiation can be found in [11]; and more recently, veriT gained proof rules to express reasoning typically used for processing, such as Skolemization, renaming of variables, and other manipulations of bound variables [2]. As veriT develops, so does the format of the proofs generated by it. There also have been efforts to improve the proof generation process. We now give an overview of the core ideas of the proofs generated by veriT before describing the concrete syntax of the proof output.

2.1 Core Concepts of the Proof Format

Assumptions. The ASSUME rule introduces a term as an assumption. The proof starts with a number of ASSUME steps. Each step corresponds to an assertion after some implicit transformations have been applied as described below. Additional assumptions can be introduced too. In this case each assumption must be discharged with an appropriate step. The only rule to do so is the SUBPROOF rule. From an assumption φ and a formula ψ proved by intermediate steps from φ , the SUBPROOF step deduces $\neg\varphi \vee \psi$ and discharges φ .

Tautologous rules and simple deduction. Most rules emitted by veriT introduce tautologies. One example is the AND_POS rule: $\neg(\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n) \vee \varphi_i$. Other rules operate on only one premise. Those rules are primarily used to simplify Boolean connectives during preprocessing. For example, the IMPLIES rule removes an implication: From $\varphi_1 \implies \varphi_2$ it deduces $\neg\varphi_1 \vee \varphi_2$.

Resolution. The proofs produced by veriT use a generalized propositional resolution rule with the rule name RESOLUTION or TH_RESOLUTION. Both names denote the same rule. The difference only serves to distinguish if the rule was introduced by the SAT solver or by a theory solver. The resolution step is purely propositional; there is currently no notion of a unifier.

The premises of a resolution step are clauses and the conclusion is a clause that has been derived from the premises by some binary resolution steps.

Quantifier Instantiation. To express quantifier instantiation, the rule FORALL_INST is used. It produces a formula of the form $(\neg\forall x_1 \dots x_n. \varphi) \vee \varphi[t_1/x_1] \dots [t_n/x_n]$, where φ is a term containing the free variables $(x_i)_{1 \leq i \leq n}$, and t_i are new variable free terms with the same sort as x_i .

The arguments of a FORALL_INST step are the list $x_1 := t_1, \dots, x_n := t_n$. While this information can be recovered from the term, providing this information explicitly aids reconstruction because the implicit transformations applied to terms (see below) obscure which terms have been used as instances. Existential quantifiers are handled by Skolemization.

Skolemization and other preprocessing steps. veriT uses the notion of a *context* to reason about bound variables. As defined above, a context is a (possibly empty) list of variables or variable term pairs. The context is modified like a stack: rules can either append elements to the right of the current context or remove elements from the right. A context Γ corresponds to a substitution σ_Γ . This substitution is recursively defined. If Γ is the empty list, then σ_Γ is the empty substitution, i.e., the identity function. If Γ is of the form Γ', x then $\sigma_\Gamma(v) = \sigma_{\Gamma'}(v)$ if $v \neq x$, otherwise $\sigma_\Gamma(v) = x$. Finally, if $\Gamma = \Gamma', x \mapsto \varphi$ then $\sigma_{\Gamma', x \mapsto \varphi} = \sigma_{\Gamma'} \circ [\varphi/x]$. Hence, the context allows one to build a substitution with the additional possibility to overwrite prior substitutions for a variable.

Contexts are processed step by step: If one step extends the context this new context is used in all subsequent steps in the step list until the context is modified again. Only a limited number of rules can be applied when the context is non-empty. All of those rules have equalities as premises and conclusion. A step with term $\varphi_1 \simeq \varphi_2$ and context Γ expresses the judgment that $\varphi_1 \sigma_\Gamma = \varphi_2$.

One typical example for a rule with context is the SKO_EX rule, which is used to express Skolemization of an existentially quantified variable. It is applied to a premise n with a context that maps a variable x to the appropriate Skolem term and produces a step m ($m > n$) where the variable is quantified.

$$\begin{array}{ccc} \Gamma, x \mapsto (\varepsilon x.\varphi) \triangleright n. & \varphi \simeq \psi & (\dots) \\ \Gamma \triangleright m. & (\exists x.\varphi) \simeq \psi & (\text{SKO_EX}; n) \end{array}$$

Example 1. To illustrate how such a rule is applied, consider the following example taken from [2]. Here the term $\neg p(\varepsilon x.\neg p(x))$ is Skolemized. The REFL rule expresses a simple tautology on the equality (reflexivity in this case), CONG is functional congruence, and SKO_FORALL works like SKO_EX, except that the choice term is $\varepsilon x.\neg\varphi$.

$$\begin{array}{ccc} x \mapsto (\varepsilon x.\neg p(x)) \triangleright 1. & x \simeq \varepsilon x.\neg p(x) & (\text{REFL}) \\ x \mapsto (\varepsilon x.\neg p(x)) \triangleright 2. & p(x) \simeq p(\varepsilon x.\neg p(x)) & (\text{CONG}; 1) \\ \triangleright 3. & (\forall x.p(x)) \simeq p(\varepsilon x.\neg p(x)) & (\text{SKO_FORALL}; 2) \\ \triangleright 4. & (\neg \forall x.p(x)) \simeq \neg p(\varepsilon x.\neg p(x)) & (\text{CONG}; 3) \end{array}$$

Linear arithmetic. Proofs for linear arithmetic use a number of straightforward rules, such as LA_TOTALITY: $t_1 \leq t_2 \vee t_2 \leq t_1$ and the main rule LA_GENERIC. The conclusion of an LA_GENERIC step is a tautology of the form $(\neg\varphi_1) \vee (\neg\varphi_2) \vee \dots \vee (\neg\varphi_n)$ where the φ_i are linear (in)equalities. Checking the validity of this formula amounts to checking the unsatisfiability of the system of linear equations $\varphi_1, \varphi_2, \dots, \varphi_n$. While Isabelle provides tactics to decide the validity of a set of linear equations, the non-trivial complexity of this task was a challenge for the proof reconstruction (see Section 3.2.4).

Example 2. The following example is the proof generated by veriT for the unsatisfiability of $(x + y < 1) \vee (3 < x)$, $x \simeq 2$, and $0 \simeq y$.

$$\begin{array}{ccc} \triangleright 1. & (3 < x) \vee (x + y < 1) & (\text{ASSUME}) \\ \triangleright 2. & x \simeq 2 & (\text{ASSUME}) \\ \triangleright 3. & 0 \simeq y & (\text{ASSUME}) \\ \triangleright 4. & \neg(3 < x) \vee \neg(x \simeq 2) & (\text{LA_GENERIC}) \\ \triangleright 5. & \neg(3 < x) & (\text{RESOLUTION}; 2, 4) \\ \triangleright 6. & x + y < 1 & (\text{RESOLUTION}; 1, 5) \\ \triangleright 7. & \neg(x + y < 1) \vee \neg(x \simeq 2) \vee \neg(0 \simeq y) & (\text{LA_GENERIC}) \\ \triangleright 8. & \perp & (\text{RESOLUTION}; 7, 6, 2, 3) \end{array}$$

```

1 (assume h1 (not (p a)))
2 (assume h2 (forall ((z1 U)) (forall ((z2 U)) (p z2))))
3 ...
4 (anchor :step t9 :args ((:= z2 vr4)))
5 (step t9.t1 (cl (= z2 vr4)) :rule refl)
6 (step t9.t2 (cl (= (p z2) (p vr4))) :rule cong :premises (t9.t1))
7 (step t9 (cl (= (forall ((z2 U)) (p z2)) (forall ((vr4 U)) (p vr4))))
8     :rule bind)
9 ...
10 (step t14 (cl (forall ((vr5 U)) (p vr5)))
11     :rule th_resolution :premises (t11 t12 t13))
12 (step t15 (cl (or (not (forall ((vr5 U)) (p vr5))) (p a)))
13     :rule forall_inst :args ((:= vr5 a)))
14 (step t16 (cl (not (forall ((vr5 U)) (p vr5))) (p a)) :rule or :premises (t15))
15 (step t17 (cl) :rule resolution :premises (t16 h1 t14))

```

Figure 1: Example proof output. Assumptions are introduced (line 1–2); a subproof renames bound variables (line 4–8); the proof finishes with instantiation and resolution steps (line 10–15)

Implicit transformations. In addition to the explicit steps, veriT performs some transformations on proof terms implicitly without creating steps. To ensure compatibility with future versions of veriT, proof reconstruction must assume that those transformations are applied between any two steps. Furthermore, veriT can not introduce additional types of implicit transformations.

- Removal of double negation: Formulas of the form $\neg(\neg\varphi)$ are silently simplified to φ .
- Removal of repeated literals: If the step formula is of the form $\varphi_1 \vee \varphi_2 \vee \dots \vee \varphi_n$ with $\varphi_i = \varphi_j$ for some $i \neq j$, then φ_j is removed. This is repeated until no more terms can be removed.
- Simplification of tautological formulas: If the step formula is of the form $\varphi_1 \vee \varphi_2 \vee \dots \vee \varphi_n$ with $\varphi_i = \top$ for some $i \neq j$, then the formula is replaced by \top .
- Reorienting equalities: veriT applies the symmetry of equality implicitly.

2.2 Concrete Syntax

The concrete text representation of the proofs generated by veriT is based on the SMT-LIB standard. Figure 1 shows an exemplary proof as printed by veriT lightly edited for readability.

We also reworked the proof syntax. Our goal is to follow the SMT-LIB standard when possible. While those modifications do not aid reconstruction inside Isabelle/HOL, they will simplify further development of the proof output. Previously, veriT produced nested steps. This was changed to a flat list of commands. The arguments of the commands are now given as annotations instead of a flat list. Since the changes are syntactical, the old format is still supported by veriT and can be selected using a command-line switch².

Figure 2 shows the grammar of the proof text generated by veriT. It is based on the SMT-LIB grammar, as defined in the SMT-LIB standard version 2.6 Appendix B³. The nonterminals $\langle \text{symbol} \rangle$,

²The option `--proof-version=N`, where N is either 1, 2, or 3.

³Available online at: <http://smtlib.cs.uiowa.edu/language.shtml>

```

    <proof> ::= <proof_command>*
    <proof_command> ::= (assume <symbol> <proof_term> )
                       | (step <symbol> <clause> :rule <symbol> <step_annotation> )
                       | (anchor :step <symbol> )
                       | (anchor :step <symbol> :args <proof_args> )
                       | (define-fun <function_def> )
    <clause> ::= (cl <proof_term>*)
    <step_annotation> ::= :premises ( <symbol>+ )
                       | :args <proof_args>
                       | :premises ( <symbol>+ ) :args <proof_args>
    <proof_args> ::= ( <proof_arg>+ )
    <proof_arg> ::= <symbol> | ( <symbol> <proof_term> )
    <proof_term> ::= <term> extended with (choice ( <sorted_var>+ ) <proof_term> )

```

Figure 2: The proof grammar

$\langle \text{function_def} \rangle$, $\langle \text{sorted_var} \rangle$, and $\langle \text{term} \rangle$ are as defined in the standard. The $\langle \text{proof_term} \rangle$ is the recursive $\langle \text{term} \rangle$ nonterminal redefined with the additional production for the `choice` binder.

Input problems in the SMT-LIB standard contain a list of *commands* that modify the internal state of the solver. In agreement with this approach veriT's proofs are also formed by a list of commands. The `assume` command introduces a new assumption. While this command could also be expressed using the `step` command with a special rule, the special semantic of an assumption justifies the presence of a dedicated command: assumptions are neither tautological nor derived from premises. The `step` command, on the other hand, introduces a derived or tautological term. Both commands `assume` and `step` require an index as the first argument to later refer back to it. This index is an arbitrary SMT-LIB symbol. The only restriction is that it must be unique for each `assume` and `step` command. The second argument is the term introduced by the command. For a `step`, this term is always a clause. To express disjunctions in SMT-LIB the `or` operator is used. Unfortunately, this operator needs at least two arguments and cannot represent unary or empty clauses. To circumvent this we introduce a new `cl` operator. It corresponds the standard `or` function extended to one argument, where it is equal to the identity, and zero arguments, where it is equal to `false`. The `:premises` annotation denotes the premises and is skipped if they are none. If the rule carries arguments, the `:args` annotation is used to denote them.

The `anchor` and `define-fun` commands are used for subproofs and sharing, respectively. The `define-fun` command corresponds exactly to the `define-fun` command of the SMT-LIB language.

2.3 Subproofs

As the name suggests, the SUBPROOF rule expresses subproofs. This is possible because its application is restricted: the assumption used as premise for the SUBPROOF step must be the assumption introduced last. Hence, the ASSUME, SUBPROOF pairs are nested. The context is manipulated in the same way: if a step pops c_1, \dots, c_n from a context Γ , there is a earlier step which pushes precisely c_1, \dots, c_n onto the context. Since contexts can only be manipulated by push and pop, context manipulations are also nested.

Because of this nesting, veriT uses the concept of subproofs. A subproof is started right before an ASSUME command or a command which pushes onto the context. We call this point the *anchor*. The subproof ends with the matching SUBPROOF command or command which pops from the context, respectively. The `:step` annotation of the anchor command is used to indicate the `step` command which will end the subproof. The term of this `step` command is the conclusion of the subproof. If the subproof uses a context, the `:args` annotation of the `anchor` command indicates the arguments added to the context for this subproof. In the example proof (Figure 1) a subproof starts on line four. It ends on line seven with the BIND steps which finished the proof for the renaming of the bound variable `z2` to `vr4`.

A further restriction applies: only the conclusion of a subproof can be used as a premise outside of the subproof. Hence, a proof checking tool can remove the steps of the subproof from memory after checking it.

2.4 Sharing and Skolem Terms

The proof output generated by veriT is generally large. One reason for this is that veriT can store terms internally much more efficiently. By utilizing a perfect sharing data structure, every term is stored in memory precisely once. When printing the proof this compact storage is unfolded.

The user of veriT can optionally activate sharing⁴ to print common subterms only once. This is realized using the standard naming mechanism of SMT-LIB. In the language of SMT-LIB it is possible to annotate every term t with a name n by writing `(! t :named n)` where n is a symbol. After a term is annotated with a name, the name can be used in place of the term. This is a purely syntactical replacement.

To limit the number of names introduced we use a simple approach: before printing the proof we iterate over all terms of the proof and recursively descend into the terms. We mark every unmarked subterm we visit. If we visit a marked term, this term gets a name. If a term already has a name, we do not descend further into this term. By doing so, we ensure that only terms that appear as child of two different parent terms get a name. Thanks to the perfect sharing representation testing if a term is marked takes constant time and the overall traversal takes linear time in the proof size.

To simplify reconstruction veriT can optionally⁵ define Skolem constants as functions. If activated, this option adds a list of `define-fun` command to define shorthand 0-ary functions for the `(choice ...)` terms needed. Without this option, no `define-fun` commands are issued and the constants are inlined.

3 Proof Reconstruction in Isabelle/HOL

Proof reconstruction is done in Isabelle in two steps presented in Figure 3: first, the proof is parsed and the terms are transformed into Isabelle terms (Section 3.1). Then we can reconstruct the proof itself by reconstructing the steps one-by-one. Some of the steps require some care (Section 3.2).

3.1 Parsing and Preprocessing

Parsing the proof is simple thanks to the infrastructure developed to reconstruct Z3 proofs for the `smt` tactic. This infrastructure is able to parse a generalized version of the SMT-LIB syntax, including the proofs generated by veriT. It produces a raw version of the proof. We only have to extract the structure

⁴By using the command-line option `--proof-with-sharing`.

⁵By using the command-line option `--proof-define-skolems`.

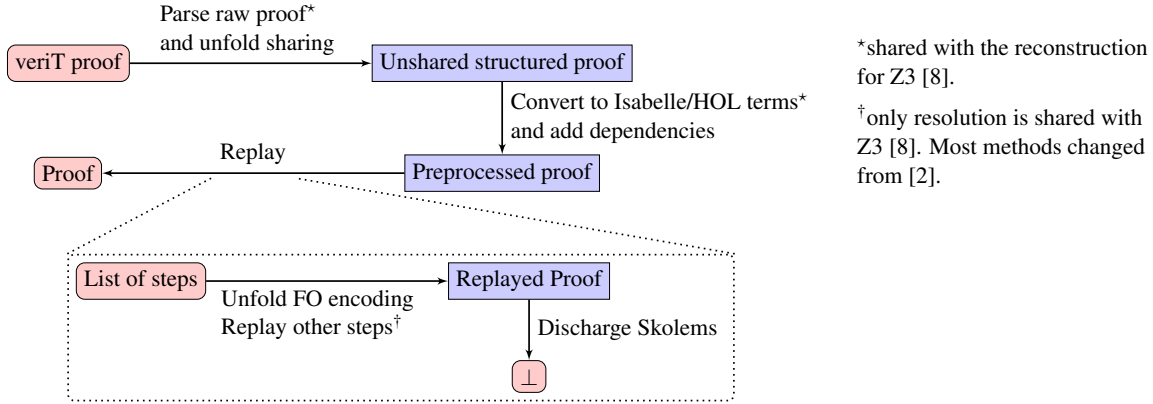


Figure 3: The reconstruction pipeline

(indices, steps, ...) from the raw proof. During parsing of the raw proof we also unfold the sharing, because Isabelle does not offer any sharing functionality.

The first transformation is a change of the disjunction representation. In the proof output, veriT represents the outermost disjunction as a multiset by using the `c1` operator. In Isabelle, we replace this multiset by a disjunction. veriT explicitly applies the rule `OR` to convert disjunctions to multisets. In Isabelle, these steps are simply the identity.

After that, we convert the SMT-LIB terms to Isabelle terms. This reuses again some of the infrastructure developed for Z3. An important difference with Z3 is the declaration of variables. When converting to Isabelle terms, types are inferred. However, some expressions like $x \mapsto y$ of the context cannot be typed without extracting the types from the conclusion.

Finally, we preprocess the proofs to ease the reconstruction further:

- We add the implicit dependency between the last step of each subproof and its conclusion. In the example of Figure 1, it is the dependency from `t9` to `t9.t2`. Spelling it out explicitly makes the reconstruction more regular.
- We add missing dependencies to the definitions of Skolem terms: veriT applies definitions implicitly, but we have to unfold the definitions explicitly to reconstruct Skolemization steps in Isabelle.

The Isabelle semantics of the proof steps are slightly different than the semantics in veriT. First, the context is seen as a list of equalities instead of a list of mappings. Second, the conclusion uses the equality symbol instead of \simeq and a substitution. If the proof step is $y \mapsto z, x \mapsto s \triangleright n. \varphi \simeq \psi$, Isabelle sees it as: $y = z, x = s \triangleright n. \varphi = \psi$, where the context are assumptions. The advantage of this different semantics is that it requires fewer transformations on the input term, as it avoids adding lambda abstractions, and makes the Isabelle tactics easier to use for reconstructions.

The difference in the semantics is small and rarely important. They only differ for variables that can syntactically appear on the left and on the right-hand side with different semantics. For example, consider $y \mapsto z, x \mapsto y \triangleright n. Pxy \simeq Pyz$. This is a tautology, because Pxy is Pyz after substitution (remember that the substitution only applies on the left-hand side). However, the naive conversion to the Isabelle version yields $y = z, x = y \triangleright n. Pxy \simeq Pyz$, which is a different term, namely $Pzz = Pzz$.

To avoid the difference in semantics, we rename terms when they have already been bound: we rename the occurrences on the right-hand side of \simeq of y by the new fresh name xy . The step $y \mapsto z, x \mapsto y \triangleright n. \varphi \simeq \psi$ becomes $y = z, x = xy \triangleright n. \varphi = \psi[xy/x]$.

3.2 Reconstructing Parsed Proofs

After parsing, we reconstruct the proof steps in Isabelle. Overall, the proof reconstruction works by replaying each step and unifying the assumptions with the premises. At the end, we get a proof of \perp .

For most steps, the rule can be spelled out as an Isabelle theorem and the only issues are implicit steps (Section 3.2.1). Unlike the reconstruction of Z3 proofs, we reconstruct subproofs as they are printed by veriT (Section 3.2.2). Finally, some rules require special care or are tricky to reconstruct: Skolemization steps, if done naively, can produce terms that are too large to be handled efficiently (Section 3.2.3); Isabelle's arithmetic procedure is incomplete and not very efficient when reconstructing arithmetic steps (Section 3.2.4); for efficiency, some rules are reconstructed heuristically (Section 3.2.5).

3.2.1 Application of Theorems

Most rules that can be applied are either tautologies or applications of theorems that can be easily expressed: The rule `TRUE` is the tautology used to prove that the theorem \top holds. Similarly, the transitivity rule `EQ_TRANSITIVE` transforms the assumptions $(t_j \simeq t_{j+1})_{j < n}$ into $t_0 \simeq t_n$.

In practice, there are two main difficulties: double negations can be simplified and equalities can be reoriented. The reorientation is implementation dependent, which prohibits us from relying on the order as given by the input problem. The reordering and simplification have consequences that either make reconstruction harder or require additional annotations in the proof output:

- Duplicate literals are implicitly removed. This is rarely an issue in practice, but we have seen this happening in some test cases like the `ITE2` rule. This rule introduces the tautology $(\text{if } \neg\varphi \text{ then } \psi_1 \text{ else } \psi_2) \vee \varphi \vee \psi_2$, but if $\varphi = \psi_2$ it produces the simplified clause $(\text{if } \neg\varphi \text{ then } \psi_1 \text{ else } \psi_2) \vee \varphi$.
- The rules can be applied up to additional negations. For example, the `ITE2` rule can be applied to get $(\text{if } \varphi \text{ then } \psi_1 \text{ else } \psi_2) \vee (\neg\varphi) \vee \psi_2$.

The individual steps are reconstructed by:

- taking into account the additional information provided in the proof output. This can require some preprocessing on the formula: in veriT instantiation (rule `FORALL_INST`) can be done to quantifiers that do not appear at the outermost level, but inside the formula. Preprocessing is used to transform $\forall x.(P \implies \neg(\exists y.Qy))$ into $\forall xy.(P \implies Qy)$. This is easier to reconstruct, because all quantifiers to instantiate are now at the outermost level and forall quantifiers.
- applying the theorem or finding the instantiations and then using `simp` to reorder the equalities and prove that the terms are equal. For the `ITE2` on $(\text{if } \neg\varphi \text{ then } \psi_1 \text{ else } \psi_2) \vee \varphi' \vee \psi_2'$, we identify the terms, φ , ψ_1 , and ψ_2 , and generate the tautology $(\text{if } \neg\varphi \text{ then } \psi_1 \text{ else } \psi_2) \vee \varphi \vee \psi_2$, that can be used by `simp` to discharge the goal by showing $\varphi = \varphi'$ and $\psi_2 = \psi_2'$. The search space is very large and the search can be very time consuming during the reconstruction.
- providing various version of the lemmas to accommodate negations: for Isabelle, the theorem $(\text{if } \neg\varphi \text{ then } \psi_1 \text{ else } \psi_2) \vee \varphi \vee \psi_2$ cannot be applied to prove $(\text{if } \varphi \text{ then } \psi_1 \text{ else } \psi_2) \vee \neg\varphi \vee \psi_2$.

In practice the reordering happens mostly when producing new terms (during parsing or instantiation). However, we do not want to rely on this specific behavior which could change in a future version.

3.2.2 Subproofs

Unlike Z3, veriT has subproofs. Subproofs fall into two categories: proofs used to justify proof steps (e.g. for Skolemization) and lemmas with assumptions and fixed variables. In Isabelle, both are modeled by the notions of contexts that encapsulate all the assumptions and fixed variables present at a given point.

The first kind of subproofs are proofs of *lemmas* that come with additional assumptions. They are used for example for proofs like $P \implies \perp$. P is an assumption of the proof (given by an ASSUME command) and \perp is the conclusion. In Isabelle, we start by extracting all the assumptions when entering the proof. This creates a new context. Then, we replay the proof in the new context. The ASSUME commands are now entailed by the context and are replayed as such. Finally, the conclusion is exported to the outer context.

Replaying *subproofs* is similar to replaying lemmas in the proof: we enter contexts with new assumptions and variables, depending on the rule. At the end of the subproof the last step is exported back to the outer context and is used to discharge the conclusion. For example, the subproof of a simple BIND step will be of the form $\forall xy. (x = y \implies Px = Qy)$ to prove that $(\forall x. Px) = (\forall y. Qy)$

3.2.3 Skolemizations

Skolemization is an important but subtle point, which slightly differs between Isabelle and veriT. While defining the constants is easy, the definitions themselves do not exactly match the natural ones and reconstructing the proof can be difficult.

Technically, Skolem constants are not introduced with a definition, but as an assumption of the form $\bar{X} = (\epsilon x. \dots)$. At the end of the reconstruction, we get the theorem $\forall \bar{X}. (\bar{X} = (\epsilon x. \neg Px) \implies \perp)$, from which we can trivially derive the theorem \perp .

Internally, veriT directly Skolemizes formulas: The term $\forall xy. Pxy$ becomes after Skolemization $P\bar{X}\bar{Y}$, i.e., $P(\epsilon x. \neg(\forall y. Pxy))(\epsilon y. \neg P(\epsilon x. \neg(\forall y. Pxy)))$, where \bar{X} and \bar{Y} are defined to be the two Skolem constants. However, in the logged proof, $\forall xy. Pxy$ becomes $\forall y. P\bar{X}y$, i.e., $\forall y. P(\epsilon x. \neg(\forall y. Pxy))y$, which in turns naturally becomes $P\bar{X}(\epsilon y. \neg P\bar{X}y)$. Therefore, in Isabelle, we fold the definition of the Skolems inside each other to get $\bar{Y} = (\epsilon y. \neg P\bar{X}y)$ and try to prove the goal. This might, however, fail due to the implicit steps. Hence, if required, we unfold all definitions and prove the result. This could explode for non-trivial terms, but we did not have issues with this during our experiments.

A major issue of the reconstruction is the size of generated terms. While developing the reconstruction, we found a case where four variables were Skolemized in a single step, and the generated term was so big that Isabelle was not able to replace the third variable by the equivalent choice: the application of the theorem $(\forall x. Px) \iff P(\epsilon x. \neg Px)$ was too slow. We now aggressively fold the Skolem constants inside the term.

3.2.4 Arithmetic

To replay arithmetic steps, we use Isabelle's procedure `linarith`. This tactic is a decision procedure for real numbers, but not for integers or natural numbers. Internally, it uses the Fourier–Motzkin elimination [17]: it derives a contradiction via a linear combination of the equations.

veriT with proof production only supports linear arithmetic. On linear problems, however, it is stronger than Isabelle's tactic: Isabelle does not simplify equations. If we have the equations $5 \times x + 10 \times y \simeq 15$, it will not be simplified to $x + 2 \times y \simeq 3$ in Isabelle. This happens neither as preprocessing, nor during the search for the linear combination. In one case over the Archive of Formal Proofs, this makes the following problem impossible to reconstruct:

$$\neg \quad 0 \leq y \quad \wedge \quad \neg \quad 10 \times x < 4 + 14 \times z \quad \wedge \\ 10 \times x \leq 15 + 25 \times y \quad \wedge \quad \neg \quad 10 \times x + 10 \times z \leq 30 + 25 \times y$$

This goal is produced as an arithmetic tautology by veriT, but `linarith` is not able to prove it. Before simplification, the inequality $16 \leq 10 \times x - 25 \times y$ is derived. After simplification, the equivalent (but

SMT calls	Number of occurrences
Successful reconstruction	447
Failed reconstruction	4
veriT timeouts	47
veriT unknown	4

Table 1: Result of using veriT instead of Z3 in existing smt calls

seemingly stronger) inequality $20 \leq 10 \times x - 25 \times y$ is derived because x and y are integers. The coefficients of the second inequality are different enough to allow `linarith` to find a contradiction, which it was unable to find otherwise.

We strengthened the reconstruction by implementing a simplification procedure that divides each equation by its greatest common divisor. It could be activated more globally, but currently conflicts with two other simplification procedures: one of them sorts terms, while the other does not.

3.2.5 Other Rules

The reconstruction of the rules is often guided by the efficiency of the reconstruction, how often a rule is used, and concrete examples. One of the most prominent rules is `CONNECTIVE_EQUIV`. It is a simplification step and can involve simplifications of the Boolean structure and arithmetic. At first, we reconstructed `CONNECTIVE_EQUIV` steps with `auto`, a tactic that simplifies the terms and performs some logical reasoning. However, this turned out to be too inefficient on large terms. Moreover, often only the Boolean structure is modified and not the terms or the order of equalities. Therefore, we now first abstract over the non-Boolean terms and check only the modifications on the Boolean structure by `fast`. Only if this fails is `auto` tried. If that also fails, `metis` is tried as a fallback tactic. We do not attempt to select the right tactic, but simply try them in this order.

4 Experimental Results

We experiment on the Isabelle reconstruction in two ways. The first one is to replace all the smt calls that are in the Isabelle distribution and are currently powered by Z3 by the version of smt with veriT. These smt calls have been selected by the developer of the library who provided the theory, because Z3 is able to find a proof and the reconstruction is fast. While this experiment provides insight into the performance of the veriT-powered smt tactic relative to the Z3-powered variant, it does not tell us if the veriT-powered one is a useful and supplementary addition to the family of automated tactics provided by Isabelle. Towards that end, we try to generate new veriT-powered smt calls by using Sledgehammer [7], an Isabelle tool able to find proofs.

4.1 Replacing the smt calls

There are already many smt calls in the theories included in the Isabelle distribution and the Archive of Formal Proofs. The latter did not allow smt calls until a few years ago. We replaced the Z3 as a backend for the smt calls, by veriT. The results are summarized in Table 1. Testing revealed:

- that veriT is not able to find all the proofs that Z3 is able to find. On the one hand, this does not corroborate the findings from the SMT competition where veriT performs better than Z3 on some

categories. On the other hand, the problems have been specifically selected to be solvable by Z3. We did not include the problems specifically relying on Z3 extensions (e.g. the division operator) or features not supported by veriT (bit vectors).

- a bug in the proof generation. veriT does not correctly print some substeps: a term is replaced by an equivalent term, but this replacement is not logged. We are currently fixing this bug in veriT. In Isabelle, this leads to an error in the reconstruction and we do not attempt to reconstruct the following steps.
- the problem in the reconstruction of arithmetic steps described in Section 3.2.4. Only one of those benchmarks could not be reconstructed without the simplification procedure.

The results are promising: we are able to reconstruct nearly all of the proofs that veriT is able to find. We cannot replace Z3 by veriT in the Isabelle distribution, but this was not the aim of this experiment.

4.2 Generating calls with Sledgehammer

Hammers, like Sledgehammer, select facts from the background theory, translate them to the input language of the provers, and then attempt to use the generated proof if a proof is found within a given timeout. The proof can be used in different ways. One approach is to gather the facts required to find a proof with one of the builtin tactics of the proof assistant. Another approach is to replay the proof within the core by an `smt`-like method or to translate it into the user-facing language of the assistant. Sledgehammer supports all of these approaches.

By default, Sledgehammer uses the following strategy: first, it tries several Isabelle tactics, including detailed proof reconstruction by the Z3-powered `smt` tactic, with a timeout of 1 s. If this is successful, it returns the tactic that was the fastest to prove the goal in Isabelle. This tactic can be inserted in the theory. If none of them is fast enough to find a proof, reconstruction of a proof in the user-facing language is attempted. We have changed Sledgehammer to additionally test the veriT-powered `smt` tactic. Sledgehammer tries to find a proof and to minimize the involved facts. Even when Z3 finds the proof, reconstruction with veriT can be faster.

We tested this approach on two formalizations: an ordered resolution prover [15, 16] and the SSA language [10, 19]. We tested all theories included in the formalization. We selected the first development because we knew that Sledgehammer was useful during the development. We selected the second formalization because it is a very different theme. Due to time constraints, we did not test more theories.

The results are given in Table 2. They show that veriT-powered `smt` calls happen in practice and can improve the speed of the overall proof processing. We do not know why veriT performs much worse on the formal SSA theory, but we believe that some rules that we do not reconstruct efficiently enough (possibly the `QNT_SIMPLIFY` rule that simplifies quantifiers) appear more often in this theory. The row ‘Oracle’ denotes calls to solvers that found a proof that could not be reconstructed. Many of these failed calls are proofs found by the SMT solver CVC4 that can either not be found or not be reconstructed by veriT and Z3-powered `smt`.

5 Related Work

Reconstruction of proofs generated by external theorem provers has been implemented in various systems including CVC in HOL Light [13], Z3 in HOL4 and Isabelle/HOL [8], and SMTCoq reconstructs veriT [1] and CVC4 [12] proofs in Coq. None of the other solvers produce detailed proofs or information on Skolemization. For veriT proofs, SMTCoq currently supports a different version of the proof output

Theory	Ordered Resolution Prover	Formal SSA
Found proofs	5019	5961
Z3-powered smt proofs	90	109
veriT-powered smt proofs	25	4
Oracle	9	63

Table 2: Proofs found by Sledgehammer on two Isabelle formalizations

(version 1) that has different rules and an older version of veriT (the version is from 2016), which does not record detailed information for Skolemization and has worse performance.

The reconstruction of Z3 proofs in HOL4 and Isabelle/HOL is one of the most advanced and well tested. It has been used to check proofs generated on problems from the SMT competition. Sadly, the code to read the SMT-LIB input problems was never included in the standard Isabelle distribution and is now lost. Proof reconstruction has been heavily tested and succeeds in more than 90% of the cases according to Sledgehammer benchmark [7, Section 9], and is very efficient.

The SMT solver CVC4 follows a different philosophy from veriT and Z3: it produces proofs in a logical framework with side conditions [18]. The output can contain programs to check certain rules. The CVC4 proof format is quite flexible but currently CVC4 does not produce proofs for quantifiers.

6 Conclusion and Future Work

We presented the syntax and semantics of the proofs generated by veriT and the reconstruction of those proofs in Isabelle. During the development, the format was extended to ease reconstruction by printing more information like the instantiations. We hope to integrate our code in the next Isabelle release.

Overall, having more details in the proofs helps to make the reconstruction more robust, because each step is simpler to check. For example, veriT detailed information on Skolemization, makes it easier to replay than the one from Z3: the reconstruction can call the ordered resolution prover `metis`. For now, the implicit simplifications prevents us from reconstructing proof more efficiently than Z3.

Another challenge is to translate the proof to the more readable Isar format. It is useful for two main reasons. First, it gives the Isabelle user more information on how the proof works and potentially what kind of lemmas would be interesting to create. Second, if the reconstruction fails, it allows the user to fix the failing part. Generating readable proofs can be done automatically by Sledgehammer for most solvers, but this does not work for veriT proofs. One reason is that, Sledgehammer does not support subproofs and inlining the assumptions each time is not very readable. Another reason is that the Skolems constants implicitly introduce a context where these constants are defined. This introduces an implicit dependency order between the definitions and every step where the defined constant appears. We could unfold the definitions to use the choice version instead, but that would harm the readability of the proof. Finally, the proofs generated by veriT often follow the scheme “ φ holds; $\varphi \leftrightarrow \psi$ also holds; hence ψ holds”, whereas “ φ hence ψ ” is easier to understand.

There are various useful pieces of information that are found by the solver but are not presented to the user. For example, in the case of linear arithmetic a contradiction is derived by finding a linear combination of the equations, but the coefficients are not printed. Therefore, Isabelle must find these same coefficients again. The reconstruction would be faster if they were in the proof output.

Acknowledgments. We thank Alex Brick, Daniel El Ouraoui, and Pascal Fontaine for suggesting many textual improvements. The work has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). Previous experiments were carried out using the Grid’5000 testbed (<https://www.grid5000.fr/>), supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several universities as well as other organizations.

References

- [1] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry & Benjamin Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*. In Jean-Pierre Jouannaud & Zhong Shao, editors: *CPP 2011, LNCS 7086*, Springer, pp. 135–150, doi:10.1007/978-3-642-25379-9_12.
- [2] Haniel Barbosa, Jasmin C. Blanchette, Mathias Fleury & Pascal Fontaine (2019): *Scalable Fine-Grained Proofs for Formula Processing*. *Journal of Automated Reasoning*, doi:10.1007/s10817-018-09502-y.
- [3] Haniel Barbosa, Jasmin C. Blanchette, Mathias Fleury, Pascal Fontaine & Hans-Jörg Schurr (2019): *Better SMT proofs for easier reconstruction*. In Thomas C. Hales, Cezary Kaliszyk, Ramana Kumar, Stephan Schulz & Josef Urban, editors: *AITP 2019*.
- [4] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2016): *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org.
- [5] Clark Barrett, Roberto Sebastiani, Sanjit Seshia & Cesare Tinelli (2009): *Satisfiability Modulo Theories*. In Armin Biere, Marijn J. H. Heule, Hans van Maaren & Toby Walsh, editors: *Handbook of Satisfiability*, chapter 26, *Frontiers in Artificial Intelligence and Applications* 185, IOS Press, pp. 825–885.
- [6] Frédéric Besson, Pascal Fontaine & Laurent Théry (2011): *A Flexible Proof Format for SMT: A Proposal*. In Pascal Fontaine & Aaron Stump, editors: *PxTP 2011*, pp. 15–26. Available at <https://hal.inria.fr/hal-00642544/>.
- [7] Jasmin C. Blanchette, Sascha Böhme, Mathias Fleury, Steffen J. Smolka & Albert Steckermeier (2016): *Semi-intelligible Isar Proofs from Machine-Generated Proofs*. *Journal of Automated Reasoning* 56(2), pp. 155–200, doi:10.1007/s10817-015-9335-3.
- [8] Sascha Böhme & Tjark Weber (2010): *Fast LCF-Style Proof Reconstruction for Z3*. In Matt Kaufmann & Lawrence C. Paulson, editors: *ITP 2010, LNCS 6172*, Springer, pp. 179–194, doi:10.1007/978-3-642-14052-5_14.
- [9] Thomas Bouton, Diego C. B. de Oliveira, David Déharbe & Pascal Fontaine (2009): *veriT: An Open, Trustable and Efficient SMT-solver*. In Renate A. Schmidt, editor: *CADE 2009, LNCS 5663*, Springer, pp. 151–156, doi:10.1007/978-3-642-02959-2_12.
- [10] Sebastian Buchwald, Denis Lohner & Sebastian Ullrich (2016): *Verified construction of static single assignment form*. In: *CC*, ACM, pp. 67–76, doi:10.1145/2892208.2892211.
- [11] David Déharbe, Pascal Fontaine & Bruno Woltzenlogel Paleo (2011): *Quantifier Inference Rules for SMT Proofs*. In Pascal Fontaine & Aaron Stump, editors: *PxTP 2011*, pp. 33–39. Available at <https://hal.inria.fr/hal-00642535>.
- [12] Burak Ekici, Guy Katz, Chantal Keller, Alain Mebsout, Andrew J. Reynolds & Cesare Tinelli (2016): *Extending SMTCoq, a Certified Checker for SMT (Extended Abstract)*. In Jasmin C. Blanchette & Cezary Kaliszyk, editors: *HaTT 2016, EPTCS 210*, pp. 21–29, doi:10.4204/EPTCS.210.5.
- [13] Sean McLaughlin, Clark Barrett & Yeting Ge (2006): *Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite*. *Electronic Notes in Theoretical Computer Science* 144(2), pp. 43–51, doi:10.1016/j.entcs.2005.12.005.
- [14] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *TACAS 2008, LNCS 4963*, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

- [15] Anders Schlichtkrull, Jasmin C. Blanchette, Dmitriy Traytel & Uwe Waldmann (2018): *Formalization of Bachmair and Ganzinger’s Ordered Resolution Prover*. *Archive of Formal Proofs*. http://isa-afp.org/entries/Ordered_Resolution_Prover.html, Formal proof development.
- [16] Anders Schlichtkrull, Jasmin C. Blanchette, Dmitriy Traytel & Uwe Waldmann (2018): *Formalizing Bachmair and Ganzinger’s Ordered Resolution Prover*. In: *IJCAR, LNCS 10900*, Springer, pp. 89–107, doi:10.1007/978-3-319-94205-6_7.
- [17] Alexander Schrijver (1999): *Theory of Linear and Integer Programming*. Wiley - Interscience Series in Discrete Mathematics and Optimization, Wiley.
- [18] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean & Cesare Tinelli (2013): *SMT Proof Checking Using a Logical Framework*. *Formal Methods in System Design* 42(1), pp. 91–118, doi:10.1007/s10703-012-0163-3.
- [19] Sebastian Ullrich & Denis Lohner (2016): *Verified Construction of Static Single Assignment Form*. *Archive of Formal Proofs*. http://isa-afp.org/entries/Formal_SSA.html, Formal proof development.

A List of Proof Rules

Rule	Description
TRUE, FALSE, AND_POS, AND_NEG, OR_POS, OR_NEG, IMPLIES_POS, IMPLIES_NEG1, IMPLIES_NEG2, EQUIV_POS1, EQUIV_POS2, EQUIV_NEG1, EQUIV_NEG2, ITE_POS1, ITE_POS2, ITE_NEG1, ITE_NEG2, EQ_REFLEXIVE, REFL, TRANS, CONG, NOT_OR, IMPLIES, NOT_IMPLIES1, NOT_IMPLIES2, EQUIV1, EQUIV2, NOT_EQUIV1, NOT_EQUIV2, ITE1, ITE2, NOT_ITE1, NOT_ITE2, ITE_INTRO	Simple rules without premises
OR, BIND, EQ_TRANSITIVE	Simple rules with premises
EQ_CONGRUENT, EQ_CONGRUENT_PRED	Congruence. Reconstruction can be problematic due to the FO encoding.
LA_RW_EQ, LA_GENERIC, LIA_GENERIC, LA_DISEQUALITY, LA_TOTALITY, LA_TAUTOLOGY	Linear arithmetics
FORALL_INST	Variable instantiation
TH_RESOLUTION, RESOLUTION	Resolution reconstructed with a simple SAT solver in Isabelle
CONNECTIVE_EQUIV	Arithmetics and logic simplification
TMP_AC_SIMP	Simplification modulo associativity and commutativity
SUBPROOF	Implication from assumption
SKO_EX, SKO_FORALL	Skolemization
QNT_SIMPLIFY, QNT_JOIN, QNT_RM_UNUSED, TMP_BFUN_ELIM	Quantifier simplification
LET, XOR1, XOR2, NOT_XOR1, NOT_XOR2, XOR_POS1, XOR_POS2, XOR_NEG1, XOR_NEG2, DISTINCT_ELIM	Unused: Isabelle does not generate XOR or lets
NLA_GENERIC, TMP_SKOLEMIZE	Unused: experimental features

CLS-SMT: Bringing Together Combinatory Logic Synthesis and Satisfiability Modulo Theories

Fadil Kallat

Tristan Schäfer

Anna Vasileva

Technical University of Dortmund,
Dortmund, Germany

{fadil.kallat, tristan.schaefer, anna.vasileva}@tu-dortmund.de

We introduce an approach that aims to combine the usage of satisfiability modulo theories (SMT) solvers with the Combinatory Logic Synthesizer (CL)S framework. (CL)S is a tool for the automatic composition of software components from a user-specified repository. The framework yields a tree grammar that contains all composed terms that comply with a target type. Type specifications for (CL)S are based on combinatory logic with intersection types. Our approach translates the tree grammar into SMT functions, which allows the consideration of additional domain-specific constraints. We demonstrate the usefulness of our approach in several experiments.

1 Introduction

In component-based software synthesis, programs are not build from scratch but composed from a repository of typed combinators. Combinators help to reduce the search space so that the inherent complexity of software synthesis problems can be handled. Moreover, additional domain-specific knowledge is contained in the semantic type layer of a repository. The underlying type system is well suited to express feature vectors of programs and software components. A user-specified repository Γ includes typed combinators that represent software components ($c : \sigma$) where c is the component name and σ is an intersection type [9, 8].

The Combinatory Logic Synthesizer (CL)S is a synthesis framework based on a type inhabitation algorithm for combinatory logic with intersection types [26, 9]. The algorithm searches for terms that are formed from the combinators and have a given target type τ . (CL)S is intended to be used for the automatic composition of software [5, 6, 9, 15, 21]. Besides the synthesis from software components, the (CL)S framework allows the synthesis of data structures, for instance of BPMN 2.0 processes [9] or planning processes [33].

Obviously, the expression of domain-specific knowledge is limited by the underlying type system. Intersection types do not explicitly take the logical connectives conjunction, disjunction and negation into consideration. Moreover, the input-output behaviour of the resulting program cannot be expressed by types. The combinatory approach allows to specify local typing information of a combinator but lacks expressivity regarding the global structure of result terms. For instance, it is not possible to state that a combinator c_0 must contain combinator c_1 anywhere in the subtree of its arguments. In some situations, not all well-formed terms might be considered to be reasonable results. Different terms might also show identical execution results and runtime behaviour.

Software synthesis is an established research topic that offers a broad range of specification formalisms such as examples [16, 17, 28, 32], types [16, 20, 25] or first-order-logic [27, 31]. For this paper, we followed the intuition that the joint usage of (complementary) formalisms can

yield a synthesis approach that combines the respective strengths of the underlying techniques. Precisely, we identified SMT to be well working with combinatory logic. There are different possible scenarios to incorporate these techniques. For example, SMT could generate parts of combinators or parametrize synthesized programs. In this paper, we show how to use SMT to filter a complete enumeration of inhabitants. We implemented our approach in a tool called CLS-SMT.

The combinatory logic synthesis yields a tree grammar that describes the set of valid inhabitants. We use this grammar to automatically construct a set of adequate SMT formulas. By solving these formulas, we receive a tree model that represents a word of the grammar. The (possibly infinite) set of inhabitants is further narrowed by introducing domain-specific structural constraints on terms. That way, we can regulate the selection of result programs while avoiding trivial solutions.

The paper is organized as follows: In Section 2 we briefly introduce the composition synthesis framework (CL)S, its underlying theoretical background and the formalism of tree grammars. Section 3 includes a presentation of CLS-SMT and the details about the translation of tree grammars into SMT formulas. In Section 4 we evaluate our approach considering an example for sort programs and a labyrinth example. Section 5 includes an overview of related work. Finally, the conclusion gives a brief summary.

2 Combinatory Logic Synthesizer (CL)S

The developing tool Combinatory Logic Synthesizer (CL)S provides an implementation of a type inhabitation algorithm for combinatory logic with intersection types that is fully integrated into the Scala programming language. The framework is publicly available [7].

The automatic software synthesis is performed by answering the type inhabitation question: $\Gamma \vdash ? : \tau$. The problem of inhabitation asks for all well-typed applicative terms that can be formed from typed combinators in a user-specified set Γ and have a given type τ . Applicative terms are defined as:

$$M, N ::= c \mid (MN)$$

A term is constructed by using named component or combinator c and application of M to N , (MN) . If there exists a combinatory expression M such that $\Gamma \vdash M : \tau$ then M is called inhabitant of τ . The type expressions that represent the specifications of term M are denoted σ, τ and are defined as follows:

$$\sigma, \tau ::= a \mid \alpha \mid \sigma \rightarrow \tau \mid \sigma \cap \tau$$

Type constants (a) can be native or semantic types. Type variables (α) are substituted with type constants and facilitate generic components. Furthermore, types can be constructed from function types ($\sigma \rightarrow \tau$) or intersections ($\sigma \cap \tau$).

There are four rules that control the type inhabitation process. According to these rules, types are assigned to combinatory terms [14]. The first rule (**var**) allows the usage of any combinator c from the typed repository Γ that has type τ using substitutions. It is defined as follows:

$$\frac{}{\Gamma, c : \tau \vdash c : \mathcal{S}(\tau)} \text{(var)}$$

Furthermore, it allows to assume that this combinator c has type $S(\tau)$, where S is a well-formed substitution on $\Gamma(c)$ mapping type variables to simple types. The inhabitation problem in general is undecidable. A restriction on variable substitution is needed to ensure decidability [14].

The following rule, arrow elimination ($\rightarrow E$), allows the application of combinators with function types to appropriately typed arguments to form terms.

$$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} (\rightarrow E)$$

The intersection introduction rule ($\cap I$), shown below, allows to type a term M with two types, if there are proofs that M has type σ and type τ .

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau} (\cap I)$$

The fourth rule (\leq) deals with subtyping.

$$\frac{\Gamma \vdash M : \sigma \quad \sigma \leq \tau}{\Gamma \vdash M : \tau} (\leq)$$

The subtyping rules are based on the Barendregt-Coppo-Dezani-Ciancaglini (BCD) [3] subtyping relation. These include for example:

$$\frac{A_2 \leq A_1 \quad B_1 \leq B_2}{A_1 \rightarrow B_1 \leq A_2 \rightarrow B_2}$$

to allow co- and contra-variant subtyping of functions and

$$\overline{A \cap B \leq A} \quad \overline{A \cap B \leq B}$$

to have intersection as the least upper bound. The BCD system is also extended with type constructors, which was proposed in [23, 10].

2.1 Tree Grammar

The (CL)S framework recursively computes all possible solutions in form of tree grammars [11]. We consider the generalized case of *normalized regular tree grammars*, which are well-known from literature [13].

Definition 1 (*Tree Grammars, Tree Grammar Languages*)

A tree grammar G is a 4-tuple $(S, \mathcal{N}, \mathcal{F}, R)$ with

- a start symbol $S \in \mathcal{N}$
- a set \mathcal{N} of nonterminals,
- a set \mathcal{F} of terminal symbols,
- a set R of productions rules of form $\alpha_1 \mapsto \{c_1(\beta_1, \beta_2, \dots, \beta_n), c_2(\gamma_1, \gamma_2, \dots, \gamma_m)\}$, where $n, m \geq 0$, $\alpha_1, \beta_1, \beta_2, \dots, \beta_n, \gamma_1, \gamma_2, \dots, \gamma_m \in \mathcal{N}$ are nonterminal and $c_1, c_2 \in \mathcal{F}$ are terminal symbols. We consider tree grammars without restriction on the arity of the terminal symbols, e.g. we can have $\alpha_1 \mapsto c_1(\beta_1, \beta_2)$ and $\alpha_2 \mapsto c_1(\beta_1)$ with $\alpha_2 \in \mathcal{N}$.

For a given tree grammar $G = (S, \mathcal{N}, \mathcal{F}, R)$ and nonterminal $\alpha \in \mathcal{N}$, $\mathcal{L}_\alpha(G)$ is the least set closed under the rule

$$\begin{aligned} & \text{if } \alpha \mapsto c(\beta_1, \beta_2, \dots, \beta_n) \in R \text{ and for all } 1 \leq k \leq n : t_k \in \mathcal{L}_{\beta_k}(G) \text{ then} \\ & c(t_1, t_2, \dots, t_n) \in \mathcal{L}_\alpha(G) \end{aligned}$$

We define $\mathcal{L}(G) = \mathcal{L}_S(G)$ to be the language of grammar G .

For request $\Gamma \vdash ? : \tau$, (CL)S constructs a tree grammar $G = (\tau, \mathcal{N}, \mathcal{F}, R)$ where $\tau \in \mathcal{N}$. The right hand sides of rules start with a combinator symbol c where $c \in \mathcal{F}$ is followed by the types of arguments required to obtain the type on the left hand side of the rule by applying the combinator. When (CL)S constructs a tree grammar, we have a word $M \in \mathcal{L}_\tau(G)$. The computed grammar G is *sound* because the word M is well-typed term. Furthermore, G is *complete* because all requested well-typed terms are words of the grammar derived for the target type τ .

2.2 Scala Implementation

The integration of the (CL)S algorithm into Scala allows simple specification of combinators [11]. A typical type specification of the repository Γ for two combinators describing a *start* position and an *up* movement in a game is

$$\begin{aligned} \Gamma = \{ & \textit{start} : \textit{Pos}(3,4), \\ & \textit{up} : (\textit{Pos}(3,4) \rightarrow \textit{Pos}(3,3)) \cap (\textit{Pos}(3,3) \rightarrow \textit{Pos}(3,2)) \}. \end{aligned}$$

Here, arrows are function types and the binary intersection type operator \cap means that a combinator has two types simultaneously. Similar to dependent types [12], specifications can include arbitrary constants and types can encode precomputed function tables. This specification mechanism is Turing complete in general [14], but in practice we use some restrictions, rendering the existence of terms for the type inhabitation problem decidable. In the current version, (CL)S accepts specifications in almost mathematical notation, allowing to state the example for Γ above as:

```
val Gamma = Map("start" -> 'Pos('3, '4),
                "up" -> ('Pos('3, '4) =>: 'Pos('3, '3)) :&:
                    ('Pos('3, '3) =>: 'Pos('3, '2)))
```

It can also extract type information from combinators with implementations attached to them, allowing to enter the combinator *up* from Γ according to the Scala representation in Listing 1. We obtain the specification with native and semantic types: $(\textit{Pos}(3,4) \rightarrow \textit{Pos}(3,3)) \cap (\textit{Pos}(3,3) \rightarrow \textit{Pos}(3,2)) \cap (\textit{Player} \rightarrow \textit{Player})$.

```
@combinator object up {
  def apply(player: Player): Player = player.goUp()
  val semanticType =
    ('Pos('3, '4) =>: 'Pos('3, '3)) :&:
    ('Pos('3, '3) =>: 'Pos('3, '2))) }
```

Listing 1: Scala representation of a combinator with native and semantic types

The intersection type operator is represented by $: \& :$ and the function types by $=>:$. The signature of `apply` is automatically translated from its native Scala type. Additional semantic type information is taken as-is and used only to impose more conditions on the use of `up`, which are user specified. The term returned for question $\Gamma \vdash ? : Pos(3,3)$ is `up(start)`, which, when providing combinator implementations, is automatically translated to the method calls `up.apply(start.apply)`. The following tree grammar is the result of the inhabitation:

$$G = \{ Pos(3,4) \mapsto \{ start() \}, \\ Pos(3,3) \mapsto \{ up(Pos(3,4)) \}, \\ Pos(3,2) \mapsto \{ up(Pos(3,3)) \} \}$$

3 CLS-SMT

This section describes the key aspects of CLS-SMT. The production rules in the grammar are used to formulate SMT constraints by using uninterpreted functions. Any SMT model satisfying the given constraints represents a tree, which is necessarily a word of the tree grammar.

We define a data structure that represents applicative terms and show how a (CL)S tree grammar can be translated to an adequate SMT formulation.

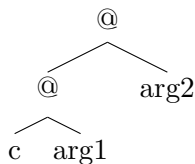
Definition 2 (*Inhabitant Tree*)

An inhabitant tree is a binary tree over integers. Let n denote the finite number of combinators used in the tree grammar and $C \subset \mathbb{N}$ range over $\{1, \dots, n\}$. With $c \in C$, an inhabitant tree is defined as follows:

$$inhabTree = 0 (leftChild inhabTree) (rightChild inhabTree) | c$$

Accordingly, the tree's alphabet of vertex labels Σ_V is $\{0\} \cup C$. A vertex labeled 0 is called application node and denoted by @. An @ node has exactly two children (i.e. 0 is a binary symbol), the function is the left child and argument is the right child. All elements of C are constants so that @ nodes are the only elements of the tree that are allowed to have children. A combinator with n arguments is represented by a tree that consists of (at least¹) n application nodes and the combinator symbol on the leftmost leaf. The n -th argument of a combinator is the right child of the combinators n -th parent. As an example, we consider the term $((c (arg1)) arg2)$, which represents the application of the binary combinator c to the arguments $arg1$ and $arg2$.

We assume that c is encoded as 1, $arg1$ as 2 and $arg2$ as 3. The corresponding inhabitant tree is $0 (leftChild (0 (leftChild 1) (rightChild 2)) (rightChild 3))$. A visual representation is as follows:



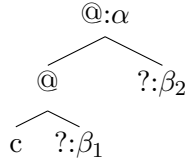
3.1 Constraint Representation

Due to the completeness of the inhabitation algorithm, there is at least one applicative term that can be build from a non-empty tree grammar. Thus, an SMT encoding of the tree grammar on its

¹more @ nodes could be contained in the subtrees representing the arguments

own will always be satisfiable. There is no need for an encoding of the subtyping relation because subtyping is considered in the inhabitation algorithm. Accordingly, the tree grammar only contains nonterminals representing types and there is a production rule for every nonterminal used.

Let V be the finite set of vertices. The labelling function $inhabitant : V \mapsto \Sigma_V$ can be used for a total representation of a tree if the rules given in Definition 2 are respected. We use the production rules in the tree grammar to formulate structural constraints on the tree. Let $n \in N$ and N denote the set of nonterminals of the grammar. We introduce the partial function $ty : V \mapsto N$, which maps vertices of a tree to a nonterminal representing a type. The information provided by a production rule of the tree grammar can now be used to systematically build constraints for the corresponding subtree. We consider the production rule $\{\alpha \mapsto \{(c(\beta_1, \beta_2))\}\}$ and its incomplete tree representation that is supplemented with the associated nonterminals:



It's possible to derive the following constraints from this production rule. Let i denote the root node of the applicative composition of the combinator and its arguments. If node i has type represented by nonterminal α then the vertex `(leftChild (leftChild i))` must be `c`, the first argument (at position `(rightChild (leftChild i))`) must be typed according to β_1 and the second argument (at `(rightChild i)`) must be typed corresponding to β_2 . The constraints for subtrees denoted by β_1 and β_2 can be formulated accordingly. Following this approach, the contents of a tree grammar can be translated into SMT constraints. Adequate assertions are formulated and supplied to the SMT solver to find implementations for the uninterpreted functions $inhabitant$ and ty . We currently use Z3 from Microsoft Research [24] to solve our formulation with the background theory LIA [4] (i.e. the linear fragment of the theory of Integers). A more detailed look at the translation will be given in the next section.

3.2 Grammar Translation

We translate the grammar by applying `TRANSLATE_PRODUCTION_RULE` shown in Algorithm 1 to every production rule of the grammar. The algorithm produces SMT boolean expressions that must evaluate to true for all vertices of a valid tree. We make use of the aforementioned functions $inhabitant$ and ty to formulate these constraints. The set of constraint functions is incorporated in an assertion with a *forall* expression where the universal quantified variable i represents the vertices. Consequently, every solution found by the SMT solver must be a word of the grammar.

Inside `TRANSLATE_PRODUCTION_RULE`, the function `Translate_Combinator` is applied to every possible combinator listed in this specific production rule. The resulting set of boolean expressions is joined with the `xor` connective as we must use one combinator subtree exclusively at a given type annotated vertex. For the sake of readability, we assume that `xor` and `and` are applicable to sets.

An n -ary combinator is translated by using the universal quantified variable i and its associated children to describe the vertices of the respective subtree. The labelling is formulated by placing constraints on the ty and $inhabitant$ functions. We reverse the list of nonterminals

args that describes the required types of a combinator's arguments in order to address the structure of inhabitant trees. That way, we can start at the root node of the current subtree and build successive address terms for each loop iteration by applying `leftChild` to the current address term. The complete structure of the subtree must satisfy all constraints that were produced in the loop, so we return the corresponding conjunction. After translating the grammar rules, we also include a root node constraint. It states that *ty* must map node 1 of the tree to the nonterminal representing the synthesis goal type.

Algorithm 1 Production Rule Translation

```

function TRANSLATE_PRODUCTION_RULE(typeId, values)
  xorSet  $\leftarrow$   $\emptyset$ 
  for all (combinator, parameters) in values do
    cTransl  $\leftarrow$  TRANSLATE_COMBINATOR(combinator, parameters)
    xorSet  $\leftarrow$  xorSet  $\cup$  cTransl
  end for
  return (ite (= (ty i typeId) (xor xorSet) true)
end function

function TRANSLATE_COMBINATOR(combinator, args)
  constrSet  $\leftarrow$   $\emptyset$ 
  currentAddress  $\leftarrow$  i
  pList  $\leftarrow$  args.reverse
  for all p in pList do
    constrSet  $\leftarrow$  constrSet  $\cup$  (= (ty (rightChild currentAddress)) p)
    constrSet  $\leftarrow$  constrSet  $\cup$  (= (inhabitant currentAddress) 0)
    currentAddress  $\leftarrow$  (leftChild currentAddress)
  end for
  combinatorConstraint  $\leftarrow$  (= (inhabitant currentAddress) combinator)
  combinedSet  $\leftarrow$  combinatorConstraint  $\cup$  constrSet
  return (and (combinedSet))
end function

```

Any tree model M^* that satisfies these constraints represents a word M of the grammar and every word M can be translated to a model M^* that satisfies these constraints. The translation is straight-forward and is thus be omitted. Let φ denote the conjunction of the constraints and τ denote the inhabitation target type, then: $M^* \models_{LIA} \varphi \Leftrightarrow M \in \mathcal{L}_\tau(G)$.

4 Experiments

In this section, we discuss the advantages and the usefulness of our approach by means of a composition of sort programs and a path finding scenario.

4.1 Sort

We consider a small repository Γ shown in Fig. 1 that can be used to compose sort programs. It contains a sort combinator for lists that applies a function to each element before performing

the sorting. The *id* combinator typed $\alpha \rightarrow \alpha$ can be used if we want to sort the unmodified list values. Moreover, the inverse function can be applied to double values. Further combinators could include the *abs* function to compare absolute values or a *dist* combinator to calculate the distance to a given value.

$$\Gamma = \{ \text{values} : \text{List}(\text{double}), \\ \text{id} : \alpha \rightarrow \alpha, \\ \text{inv} : \text{double} \rightarrow \text{double}, \\ \text{sortmap} : (\alpha \rightarrow \alpha) \rightarrow \text{List}(\alpha) \rightarrow \text{SortedList}(\alpha), \\ \text{min} : \text{double} \rightarrow \text{SortedList}(\text{double}) \rightarrow \text{minimal} \cap \text{double}, \\ \text{default} : \text{double} \}$$

Figure 1: Repository for the sort example

In some cases, it might be required to sort a *double* list and additionally determine its minimal value. The corresponding combinator *min* will be implemented by extracting the first value of a sorted list (assuming that we always sort in an ascending order). The result type of *min* is an intersection of *minimal* and *double*. For empty lists, a default value will be returned. In this example, such a value is held in the component *default*, which has the type *double*. The inhabitation request $\Gamma \vdash ? : \text{minimal} \cap \text{double}$ yields the following grammar *G*:

$$G = \{ \text{SortedList}(\text{double}) \mapsto \{ \text{sortmap}(\text{double} \rightarrow \text{double}, \text{List}(\text{double})) \}, \\ \text{minimal} \cap \text{double} \mapsto \{ \text{id}(\text{minimal} \cap \text{double}), \text{min}(\text{double}, \text{SortedList}(\text{double})) \}, \\ \text{double} \mapsto \{ \text{id}(\text{double}), \text{default}(), \text{inv}(\text{double}), \text{min}(\text{double}, \text{SortedList}(\text{double})) \}, \\ \text{double} \rightarrow \text{double} \mapsto \{ \text{id}(), \text{inv}() \} \\ \text{List}(\text{double}) \mapsto \{ \text{id}(\text{List}(\text{double})), \text{values}() \} \}$$

Figure 2: Tree grammar for the sort example, $\Gamma \vdash ? : \text{minimal} \cap \text{double}$

A double value can be formed by applying *id* or *inv* to any term with type double. Obviously, terms like *inv* and *id* can be applied an arbitrary number of times to arguments of type double. Thus, the range of terms with type double is infinite. Moreover, a term typed *minimal* \cap *double* can also be used as the first argument of the *min* operator. The grammar describes all well-formed solutions that comply to the target type. However, it is clearly not desirable to compose infinite range of trivial solutions. With extensions formulated as SMT constraints, we can further filter the result set without specializing Γ too much.

In order to avoid trivial solutions, we specify *id* and *inv* to be used only as arguments. Moreover, the first argument of *min* must be a terminal. Given the indices 2, 3 and 5 for the combinators *id*, *min* and *inv*, the following assertions are added to the SMT script:

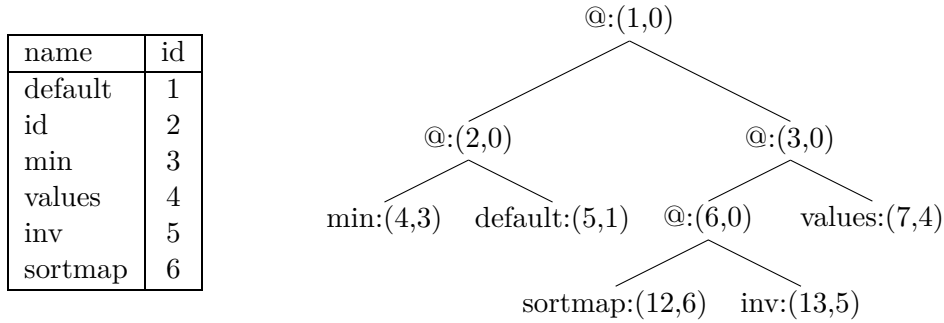
```
(assert (forall ((i Int)) (not (= (inhabitant (leftChild i)) 2))))
(assert (forall ((i Int)) (not (= (inhabitant (leftChild i)) 5))))
(assert (forall ((i Int))
  (ite (= (inhabitant (leftChild i)) 3)
```


$(\text{not } (= (\text{inhabitant } (\text{rightChild } i)) 0)) \text{ true}))$

With these constraints at hand, only two valid solutions are found for the inhabitation request $\Gamma \vdash ? : \text{minimal} \cap \text{double}$:

$((\text{min default}) ((\text{sortmap inv}) \text{values}))$ and
 $((\text{min default}) ((\text{sortmap id}) \text{values}))$

The combinator *min* is applied to the terms yielded by the combinators *default* and *sortmap*. For this particular example, the combinator mapping in the table shown below was used. In order to illustrate the first result term as a tree, we use the following labelling pattern: *combinator name* : (*vertex id*, *combinator id*)



4.2 Labyrinth Example

In the following labyrinth example, it is possible to go *up*, *down*, *left* or *right*, if the new position is not occupied by obstacles [11]. Fig. 3 illustrates a 3×4 labyrinth example. The starting position is $(0,2)$ (shown as \bullet) and the goal position $(1,0)$ (shown as \star).

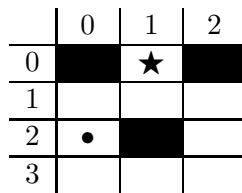


Figure 3: Labyrinth example

The repository with typed combinators for this example is represented in Fig. 4.

$$\begin{aligned}
\Gamma_{Lab} = \{ & \textit{left} : (Pos(1,1) \rightarrow Pos(0,1)) \cap Pos(2,1) \rightarrow Pos(1,1) \cap \\
& (Pos(1,3) \rightarrow Pos(0,3)) \cap (Pos(2,3) \rightarrow Pos(1,3)), \\
& \textit{right} : (Pos(0,1) \rightarrow Pos(1,1)) \cap (Pos(1,1) \rightarrow Pos(2,1)) \cap \\
& (Pos(0,3) \rightarrow Pos(1,3)) \cap (Pos(1,3) \rightarrow Pos(2,3)), \\
& \textit{up} : (Pos(0,3) \rightarrow Pos(0,2)) \cap (Pos(2,3) \rightarrow Pos(2,2)) \cap \\
& (Pos(1,1) \rightarrow Pos(1,0)) \cap (Pos(0,2) \rightarrow Pos(0,1)) \cap \\
& (Pos(2,2) \rightarrow Pos(2,1)), \\
& \textit{down} : (Pos(1,0) \rightarrow Pos(1,1)) \cap (Pos(0,1) \rightarrow Pos(0,2)) \cap \\
& (Pos(2,1) \rightarrow Pos(2,2)) \cap (Pos(0,2) \rightarrow Pos(0,3)) \cap \\
& (Pos(2,2) \rightarrow Pos(2,3)), \\
& \textit{start} : Pos(0,2) \}
\end{aligned}$$

Figure 4: Repository for the labyrinth example shown in Fig. 3

The combinators *up*, *down*, *left*, and *right* can be used to go from position $Pos(x,y)$ to an accessible neighbouring position. The types $Pos(x,y)$ represent the column and row positions. For example, combinator *left* can be used to go from position $Pos(1,1)$ to position $Pos(0,1)$ as well as from $Pos(2,1)$ to $Pos(1,1)$, from $Pos(1,3)$ to $Pos(0,3)$, and from $Pos(2,3)$ to $Pos(1,3)$. The combinator *start* provides the starting position.

To get all possible paths from start (0,2) to goal position (1,0), we ask for:

$$\Gamma \vdash ? : Pos(1,0)$$

For this goal position the algorithm computes the grammar shown in Fig. 5.

$$\begin{aligned}
G = \{ & Pos(1,0) \mapsto \{up(Pos(1,1))\}, \\
& Pos(1,1) \mapsto \{right(Pos(0,1)), left(Pos(2,1)), down(Pos(1,0))\}, \\
& Pos(1,1) \mapsto \{up(Pos(0,2)), left(Pos(1,1))\}, \\
& Pos(2,1) \mapsto \{up(Pos(2,2)), right(Pos(1,1))\}, \\
& Pos(2,2) \mapsto \{down(Pos(2,1)), up(Pos(2,3))\}, \\
& Pos(0,1) \mapsto \{up(Pos(0,2)), left(Pos(1,1))\}, \\
& Pos(0,3) \mapsto \{down(Pos(0,2)), left(Pos(1,3))\}, \\
& Pos(0,2) \mapsto \{down(Pos(0,1)), up(Pos(0,3)), start()\}, \\
& Pos(1,3) \mapsto \{left(Pos(2,3)), right(Pos(0,3))\}, \\
& Pos(2,3) \mapsto \{down(Pos(2,2)), right(Pos(1,3))\} \}
\end{aligned}$$

Figure 5: Tree grammar for the labyrinth example

For the path going *up*, *right*, and *up* the algorithm constructs a term $up(right(up(start)))$. In this example, there are also terms that represent trivial paths with cycles. For example:

$$\begin{aligned}
& up(right(up(down(up(down(up(start))))))), \\
& down(up(up(right(up(start))))), \dots
\end{aligned}$$

By means of SMT solvers, we can restrict the number of solutions computed by (CL)S in order to avoid trivial terms. For example, we can decide, which combinators have to be used and how often. As presented in Section 3 we translate the computed tree grammar (s. Fig. 5) to SMT expressions by means of algorithm 1.

In order to filter the inhabitants, we consider domain-specific constraints. We are able to select, which combinators should be used in the solution. For instance, Fig. 6 shows a formula that states a term should not include combinator *down* (translated as `(= (inhabitant i) 1)`). This way, we constrain the usage of certain combinator. In this particular example (see Fig. 3), we might want to avoid the *down* combinator, because the robot has to get to the top-right goal position.

```
(assert (forall ((i Int)) (not (= (inhabitant i) 1))))
```

Figure 6: Assertion for filtering of combinator

We reduce the number of cycles and define the order of usage of the combinators in order to avoid unnecessary paths. For example, we can formulate a constraint that forbids the application of combinator *down* (index 1) to combinator *up* (index 2) and vice versa. The same applies to combinators *left* (index 3) and *right* (index 4). Fig. 7 shows the definition of this rule.

```
(assert (forall ((i Int))
  (and
    (not (and (= (inhabitant (leftChild i)) 3)
              (= (inhabitant (leftChild (rightChild i))) 4)))
    (not (and (= (inhabitant (leftChild i)) 4)
              (= (inhabitant (leftChild (rightChild i))) 3)))
    (not (and (= (inhabitant (leftChild i)) 2)
              (= (inhabitant (leftChild (rightChild i))) 1)))
    (not (and (= (inhabitant (leftChild i)) 1)
              (= (inhabitant (leftChild (rightChild i))) 2))))
  ))
```

Figure 7: Formula for definition of order

5 Related Work

Type-theoretical specification

There are various approaches to solve synthesis problems by means of type theory. For instance, Polikarpova et al. synthesized recursive functions satisfying a specification in the form of polymorphic refinement types [25]. Zdancewic et al. demonstrated that examples in example-directed synthesis can be interpreted as refinement types [16]. They provided an example-based specification language by using intersection types with singletons. In contrast, (CL)S expresses semantic specifications with intersection types. Kuncak et al. used type inhabitation in the

simply typed lambda calculus to support developers by generating a list of valid expressions of a given type for code completion [20].

SMT

In the last decades, there have been many approaches using SMT solvers for synthesis. A common property of those methodologies is the use of syntactic constraints and a correctness specification. In 2006, preliminary work in template-based synthesis was undertaken by Solar-Lezama et al. [29]. In *Sketching*, a partial implementation is given and synthesis completes missing parts by considering a specification of the desired functionality [29]. Following this idea, loop-free bitvector programs [18] and deobfuscating programs [22] were synthesized in a component-based manner. In contrast to our work, desired functionality and components were specified as logical relations between the input and output variables [18, 22]. Another approach in SMT based synthesis is programming by examples. A user specifies the behaviour of the desired program by a number of input-output examples [19]. Singh and Gulwani transformed strings and data types in spreadsheets [17, 28] and Udupa et al. were able to synthesize protocols from a given skeleton and examples [32].

In 2013, a number of researchers picked up the main ideas of the projects above to formulate the problem of syntax-guided synthesis (SyGuS) [1]. The Counterexample-Guided Inductive Synthesis (CEGIS) architecture describes how SyGuS problems can be tackled by learning from counterexamples provided by a verification oracle, which is often implemented by off-the-shelf SMT solvers [1].

Most of the synthesis algorithms based on CEGIS variants are solving $\exists\forall$ -formulas iteratively using SMT solvers [1]. Similar to Reynolds et al. we consider synthesis as a theorem-proving problem. In our case, the problem is solved in combinatory logic and later refined by a SMT solver, whereas in [27] the problem is solely solved within the SMT solver. The main difference is the way of specification. Like in many traditional synthesis approaches [16, 17, 28, 32], targets in [27] are specified by using properties of executed programs. More specifically, relations on inputs and outputs are defined. This allows for a fine-granular specification on program behaviour, but it is hard to control the structure of synthesized programs. It can also be hard to specify the program behaviour in the SMT solver, which becomes especially apparent in the presence of side effects or exceptions. In (CL)S, these concerns are hidden behind the interfaces of types. Types are particularly easy to define, because they already exist in most programming languages and do not need to be specified just for synthesis. They can encode taxonomic concepts via semantic types and subtyping, which is usually a very natural way of expression [30]. In future work, it might be interesting to consider bridging the gap between behavioural and type-based specifications. In particular, the approach in [27] could be used to synthesize the implementation for individual combinators, which are then composed by (CL)S and CLS-SMT.

6 Conclusion

In our work we combined Combinatory Logic Synthesis and Satisfiability Modulo Theories in a tool called CLS-SMT. In this way, we are able to compensate limitations of one technology by taking advantage of the other and vice versa. The synthesis framework (CL)S generates a tree grammar from a given repository of typed components that contains domain-specific knowledge. We should emphasize that the tree grammar is *complete* and describes all well-formed solutions.

CLS-SMT translates the grammar into SMT formulas and further domain-specific constraints are added. The SMT solver Z3 finds a model considering the translated tree grammar and constraints.

By having further constraints formulated as SMT formulas, we are able to restrict inhabitants without restricting the types of the (CL)S component repository. That way, we benefit from the expressiveness of first-order logic and background theories. Combinatory logic synthesis reduces the search space of the SMT solver. In general, SMT considers this structure of the programs, whereas components in (CL)S contain domain-specific details. Combinatory logic with intersection types is a Turing complete formalism that allows to define semantic taxonomies based on subtyping [9].

Although SMT solvers are highly efficient through decades of research and improvements, handling quantified formulas is still challenging. Congruence Closure with Free Variables (CCFV) [2] is a framework that is based on the E-ground (dis)unification problem and unifies major instantiation techniques in SMT solving. Experimental evaluation shows that CCFV improved the performance of the solvers CVC4 and veriT significantly, so that the former outranks the state-of-the-art in instantiation based SMT solving. Within our research, the replacement of solvers is possible with reasonable effort due to the SMT-LIB standard. Further performance enhancements could be achieved by exploring the usage of data types to express the tree.

We have applied CLS-SMT to synthesize sort programs and motion plans. Motion planning problems are an interesting topic for program synthesis because of the associated scaling problems. Our examination shows that synthesis of small motion plans is successful. On the other hand, we found that larger examples do not scale properly. Our approach is well-suited for motion plans with up to 10×10 tiles. Scaling problems do not apply to other use cases such as the sort example. Future work considers an investigation of motion planning problems with multiple robot instances and obstacles.

Acknowledgement. The work presented in this paper was partly funded by the GRK 2193 (www.grk2193.tu-dortmund.de/de/) and the Center of Excellence for Logistics and IT (www.leistungszentrum-logistik-it.de/) located in Dortmund.

References

- [1] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak & A. Udupa (2013): *Syntax-guided synthesis*. In: *2013 Formal Methods in Computer-Aided Design*, pp. 1–8, doi:10.1109/FMCAD.2013.6679385.
- [2] Haniel Barbosa, Pascal Fontaine & Andrew Reynolds (2017): *Congruence Closure with Free Variables*. In Axel Legay & Tiziana Margaria, editors: *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science* 10206, Springer, Berlin, Heidelberg, pp. 214–230, doi:10.1007/978-3-662-54580-5_13.
- [3] H. P. Barendregt, M. Coppo & M. Dezani-Ciancaglini (1983): *A Filter Lambda Model and the Completeness of Type Assignment*. *Journal of Symbolic Logic* 48(4), pp. 931–940, doi:10.2307/2273659.
- [4] Barrett, C., Fontaine, P., Tinelli, C.: *SMT-LIB Logics*. Available at <http://smtlib.cs.uiowa.edu/logics.shtml>.
- [5] Jan Bessai, Tzu-Chun Chen, Andrej Dudenhefner, Boris Döder, Ugo de'Liguoro & Jakob Rehof (2018): *Mixin Composition Synthesis based on Intersection Types*. *Logical Methods in Computer Science* Volume 14, Issue 1, doi:10.23638/LMCS-14(1:18)2018.

- [6] Jan Bessai, Boris Döder, George T. Heineman & Jakob Rehof (2015): *Combinatory Synthesis of Classes Using Feature Grammars*. In: *Revised selected papers of the 12th International Conference on Formal Aspects of Component Software*, pp. 123–140, doi:10.1007/978-3-319-28934-2_7.
- [7] Jan Bessai, Boris Döder, Geroge T. Heineman et al. (2018): *(CL)S Framework*. Available at <http://www.combinators.org>. Accessed: 2018-04-30.
- [8] Jan Bessai, Andrej Dudenhefner, Boris Döder, Moritz Martens & Jakob Rehof (2014): *Combinatory Logic Synthesizer*. In Tiziana Margaria & Bernhard Steffen, editors: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, Lecture Notes in Computer Science 8802*, Springer, Berlin, Heidelberg, pp. 26–40, doi:10.1007/978-3-662-45234-9_3.
- [9] Jan Bessai, Andrej Dudenhefner, Boris Döder, Moritz Martens & Jakob Rehof (2016): *Combinatory Process Synthesis*. In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, pp. 266–281, doi:10.1007/978-3-319-47166-2_19.
- [10] Jan Bessai, Jakob Rehof & Boris Döder (2019): *Fast Verified BCD Subtyping*. In: *Models, Mindsets, Meta: The What, the How, and the Why Not?, Lecture Notes in Computer Science 11200*, Springer, Cham, [S.l.], pp. 356–371, doi:10.1007/978-3-030-22348-9_21.
- [11] Jan Bessai & Anna Vasileva (2018): *User Support for the Combinator Logic Synthesizer Framework*. *Electronic Proceedings in Theoretical Computer Science* 284, pp. 16–25, doi:10.4204/EPTCS.284.2.
- [12] Edwin Brady (2017): *Type-driven development with Idris*. Manning Publications Co, Shelter Island, NY.
- [13] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison & M. Tommasi (2007): *Tree Automata Techniques and Applications*. Available online: <http://www.grappa.univ-lille3.fr/tata>. Release October, 12th 2007.
- [14] Boris Döder, Moritz Martens, Jakob Rehof & Paweł Urzyczyn (2012): *Bounded Combinatory Logic*. In Patrick Cégielski & Arnaud Durand, editors: *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France, LIPIcs 16*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 243–258, doi:10.4230/LIPIcs.CSL.2012.243.
- [15] Boris Döder, Jakob Rehof & George T. Heineman (2015): *Synthesizing type-safe compositions in feature oriented software designs using staged composition*. In: *Proceedings of the 19th International Conference on Software Product Lines*, pp. 398–401, doi:10.1145/2791060.2793677.
- [16] Jonathan Frankle, Peter-Michael Osera, David Walker & Steve Zdancewic (2016): *Example-directed Synthesis: A Type-theoretic Interpretation*. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16*, ACM, New York, NY, USA, pp. 802–815, doi:10.1145/2837614.2837629.
- [17] Sumit Gulwani, William R. Harris & Rishabh Singh (2012): *Spreadsheet data manipulation using examples*. *Communications of the ACM* 55(8), pp. 97–105, doi:10.1145/2240236.2240260.
- [18] Sumit Gulwani, Susmit Jha, Ashish Tiwari & Ramarathnam Venkatesan (2011): *Synthesis of Loop-free Programs*. In: *Proceedings of PLDI'11*, ACM, p. 62, doi:10.1145/1993498.1993506.
- [19] Sumit Gulwani, Oleksandr Polozov & Rishabh Singh (2017): *Program Synthesis. Foundations and Trends® in Programming Languages* 4(1-2), pp. 1–119, doi:10.1561/2500000010.
- [20] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj & Ruzica Piskac (2013): *Complete Completion Using Types and Weights*. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, ACM, New York, NY, USA, pp. 27–38, doi:10.1145/2491956.2462192.

- [21] George T. Heineman, Jan Bessai, Boris Döder & Jakob Rehof (2016): *A Long and Winding Road Towards Modular Synthesis*. In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, pp. 303–317, doi:10.1007/978-3-319-47166-2_21.
- [22] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia & Ashish Tiwari (2010): *Oracle-guided component-based program synthesis*. In Jeff Kramer, Judith Bishop, Prem Devanbu & Sebastian Uchitel, editors: *ACM/IEEE 32nd International Conference on Software Engineering, 2010*, IEEE, NJ, USA, p. 215, doi:10.1145/1806799.1806833.
- [23] Olivier Laurent (2018): *Intersection Subtyping with Constructors*. In Michele Pagani & Sandra Alves, editors: *Proceedings DCM 2018 and ITRS 2018, EPTCS 293*, Oxford, UK, pp. 73–84, doi:10.4204/EPTCS.293.6.
- [24] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science 4963*, Springer, Berlin, Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.
- [25] Nadia Polikarpova, Ivan Kuraj & Armando Solar-Lezama (2016): *Program Synthesis from Polymorphic Refinement Types*. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '16*, ACM, New York, NY, USA, pp. 522–538, doi:10.1145/2908080.2908093.
- [26] Jakob Rehof (2013): *Towards Combinatory Logic Synthesis*. In: *BEAT 2013, 1st International Workshop on Behavioural Types*, ACM.
- [27] Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark Barrett & Morgan Deters (2017): *Refutation-based synthesis in SMT*. *Formal Methods in System Design*, doi:10.1007/s10703-017-0270-2.
- [28] Rishabh Singh & Sumit Gulwani (2016): *Transforming spreadsheet data types using examples*. In Rastislav Bodik & Rupak Majumdar, editors: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016*, ACM, New York, NY, USA, pp. 343–356, doi:10.1145/2837614.2837668.
- [29] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia & Vijay Saraswat (2006): *Combinatorial sketching for finite programs*. In John Paul Shen, editor: *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ACM, New York, NY, USA, p. 404, doi:10.1145/1168857.1168907.
- [30] Bernhard Steffen, Tiziana Margaria & Michael von der Beeck (1997): *Automatic Synthesis of Linear Process Models from Temporal Constraints: An Incremental Approach*. In: *ACM/SIGPLAN Int. Workshop on Automated Analysis of Software (AAS'97)*.
- [31] Ashish Tiwari, Adrià Gascón & Bruno Dutertre (2015): *Program Synthesis Using Dual Interpretation*. In Amy Felty & Aart Middeldorp, editors: *Automated deduction – CADE-25, LNCS sublibrary. SL 7, Artificial intelligence 9195*, Springer, Cham, pp. 482–497, doi:10.1007/978-3-319-21401-6_33.
- [32] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin & Rajeev Alur (2013): *TRANSIT: Specifying Protocols with Concolic Snippets*. *SIGPLAN Not.* 48(6), pp. 287–296, doi:10.1145/2499370.2462174.
- [33] Jan Winkels, Julian Graefenstein, Tristan Schäfer, David Scholz, Jakob Rehof & Michael Henke (2018): *Automatic Composition of Rough Solution Possibilities in the Target Planning of Factory Planning Projects by Means of Combinatory Logic*. In Tiziana Margaria & Bernhard Steffen, editors: *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice, Lecture Notes in Computer Science 11247*, Springer, Cham, pp. 487–503, doi:10.1007/978-3-030-03427-6_36.