

A Standardized Syntax for Connection Proofs

Jens Otten¹, Sean B. Holden²

¹*Department of Informatics, University of Oslo, Norway*

²*Department of Computer Science and Technology, University of Cambridge, United Kingdom*

Abstract

Providing a proof certificate is one of the most important features of (fully automated) theorem proving systems. For theorem provers based on, e.g. resolution or superposition calculi, a syntax for writing solutions is well documented and used. Even though attempts have been made to specify a syntax for connection calculi, there is currently no such syntax in common use. This paper provides an overview of the proof syntax used by existing connection provers and proposes a standardized syntax for connection proofs based on the TPTP language. Due to the general approach, the syntax can be extended to also represent connection proofs in non-classical logics.

Keywords

Automated Reasoning, logic, connection calculus, connection proofs, syntax, TPTP, proof certificate

1. Introduction

One of the most important features of automated theorem proving (ATP) systems is the output of a proof certificate, which contains a detailed description of the proof. Such a proof certificate can be used to verify the output given by an ATP system. It also increases the interoperability between ATP systems, ATP tools, and application software. Such ATP tools can, for example, be used to represent found proofs in a more readable form.

The writing of derivations in resolution calculi using the TPTP syntax is well documented and specified [19]. For many other calculi, such as connection or sequent calculi, such a common specification does not exist. Derivations in these calculi differ significantly from derivations in the resolution calculus. Whereas the leaves of a proof in the connection calculus consists of the axioms of the *calculus*, the leaves of a derivation in the resolution calculus consists of the axiom formulae of the given *problem*.

This paper proposes a syntax for connection proofs. It should not be seen as a final specification, but as a foundation to initiate discussions within the community in order to finalize such a specification. We start with a description of the requirements for a standardized syntax (Section 2), before presenting details of existing syntaxes for proofs, including the well-known TPTP syntax and the syntaxes used by the connection provers leanCoP and CONNECT++ (Section 3). Afterwards, we specify our proposed syntax for connection proofs (Section 4), before concluding with a summary and outlook (Section 5).

AReCCa 2023: Automated Reasoning with Connection Calculi, 18 September 2023, Prague, Czech Republic


✉ jeotten@ifi.uio.no (J. Otten); sbh11@cl.cam.ac.uk (S. B. Holden)

🌐 <http://jens-otten.de/> (J. Otten); <http://www.cl.cam.ac.uk/~sbh11> (S. B. Holden)

🆔 0000-0002-4331-8698 (J. Otten); 0000-0001-7979-1148 (S. B. Holden)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

2. Preliminaries

We describe requirements for a proof syntax and introduce the formal connection calculus.

2.1. Requirements for a Proof Syntax

There are certain requirements that a proof syntax should fulfill in order to be widely accepted by the community. These are:

- *Simple/lean*: The syntax should be as concise and simple as possible allowing a natural and straightforward representation of proofs. A proof syntax that is appropriate for one calculus might not be suitable to represent proofs in another calculus.
- *Human-readable*: Even though one of the main purposes of a standardized proof syntax is the interoperability between ATP systems, ATP tools and application software, a proof in such a syntax should be readable by a human.
- *Based on established formats*: The proof syntax should consider the syntax of previous proof formats, whenever these can be used in a straightforward way and do not make the proof representation more complicated than necessary (see first requirement).
- *Well documented and specified*: The proof syntax should be described in a concise and unambiguous way.
- *Extendable*: The proof syntax should be kept general enough to be extendable to similar or related proof calculi, for example for non-classical logics.
- *Efficiently verifiable*: It should be possible to verify the correctness of a “proof” given in the specified syntax in polynomial time.

In general, a *proof* of (the validity of) a first-order formula F is a proof of F in a (correct) proof calculus. We will focus on the clausal connection calculus, which we will define now.

2.2. The Connection Calculus

Connection calculi, including the connection method [2], the connection tableau calculus [8] and the model elimination calculus [9], are established proof search calculi. We use the standard language of *classical first-order logic*. A (*first-order*) *formula* (denoted by F) is built up from atomic formulae, denoted by A , the connectives \neg , \wedge , \vee , \Rightarrow , and the first-order quantifiers \forall and \exists . A *literal* L has the form A or $\neg A$.

In the *clausal connection calculus* [2, 14] a formula is represented as a matrix. The (*classical*) *matrix* $M(F)$ of a formula F is its representation as a set of clauses, where each clause is a set of literals. It is the representation of F in *disjunctive normal form*. Skolemization of the Eigenvariables is done in the usual way. In the *graphical representation* of a matrix, its clauses are arranged horizontally, while the literals of each clause are arranged vertically.

In contrast to sequent and tableau calculi, the proof search in connection calculi is guided by connections. A *connection* is a set $\{A_1, \neg A_2\}$ of literals with the same atomic formula, but different “polarities”. A *term substitution* σ assigns terms to variables. A connection $\{A_1, \neg A_2\}$ is σ -*complementary* iff $\sigma(A_1) = \sigma(A_2)$.

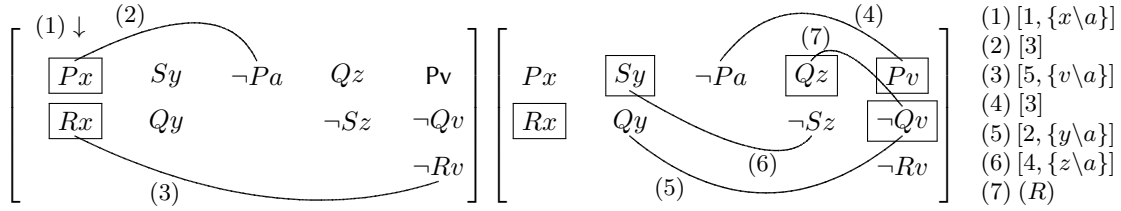


Figure 2: A proof for M_a using the graphical matrix representation.

3. Existing Syntaxes for Connection Proofs

We present the syntax used for the TPTP library and by the provers leanCoP and CONNECT++.

3.1. The TPTP Syntax

The TPTP language is suitable for representing problems as well as derivations in first-order and higher-order logic [18, 19]. The top level building blocks are of the following form:

language(*name*, *role*, *formula*, *source*, *useful_info*).

language is, for example, *fof* or *cnf*, for first-order or clause normal form. Each formula has a unique *name*. *role* is a label such as *axiom* or *conjecture*. The *source* describes where the formula came from, for example an input file, and *useful_info* is a list of user information. The last two fields are optional.

Example 2. Pelletier’s original problem 24 is in the TPTP library under the name SYN054+1. The representation of its simplified problem 24a from Example 1 in TPTP syntax is given in Figure 3.

A *derivation* or *proof* written in the TPTP language is a list of annotated formulae, as for problems. For derivations/proofs the *source* has one of the forms

file(*file_name*,*file_info*)
inference(*inference_name*,*inference_info*,*parents*)

The former is used for formulae taken from the problem file. The latter is used for inferred formulae, in which *inference_name* is the name of the inference rule, *inference_info* is a list of additional information about the inference, and *parents* is a list of the parents’ node names in the derivation. The *inference_info* contains items such as variable bindings captured within *bind/2* terms.

```

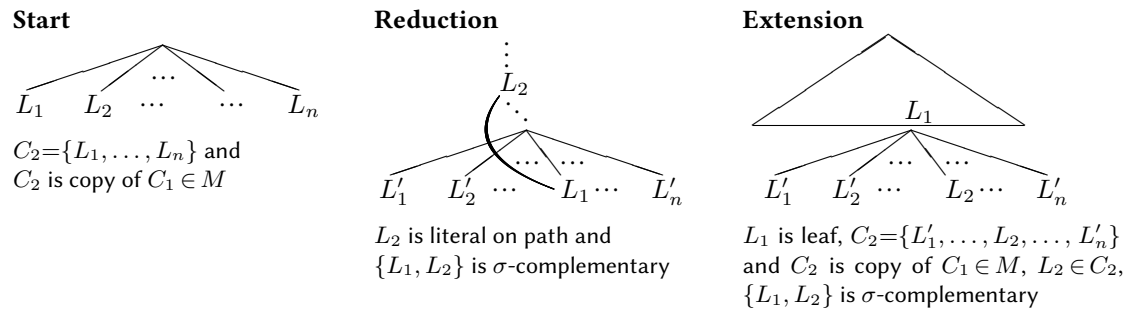
%-----
fof(pel24,conjecture, ( ? [X] : ( big_p(X) & big_r(X) ) )).
fof(pel24_1,axiom, ( ~ ( ? [X] : ( big_s(X) & big_q(X) ) ) )).
fof(pel24_2,axiom, ( ! [X] : ( big_p(X) => ( big_q(X) | big_r(X) ) ) )).
fof(pel24_3a,axiom, ( ? [X] : big_p(X) )).
fof(pel24_4a,axiom, ( ! [X] : ( big_q(X) => big_s(X) ) )).
%-----

```

Figure 3: The presentation of Pelletier’s problem 24a in TPTP syntax, called SYN054a.

Table 1

The rules of the connection calculus using the connection tableau representation.



3.2. The leanCoP Syntax

According to [19], a refutation (or proof) is a derivation that has the root node *false*, representing the empty clause. Whereas this description fits proofs in resolution calculi, it is not directly applicable to proofs in sequent, tableau or connection calculi. An attempt to “squeeze” connection proofs into this resolution-style frame [15] resulted in a syntax that does not conform to the first two requirements suggested in Section 2.1: readability and simplicity. For this reason, leanCoP [14, 11] uses a syntax, called *leanTPTP*, that represents connection proofs in a more natural way, while still using main components of the TPTP syntax.

In order to shorten the output, leanCoP returns connection proofs in the connection tableau representation [8]. The rules of the *connection tableau calculus* are shown in Table 1 and correspond to the rules of the formal calculus in Figure 1. There is no explicit *Axiom*, which is applied exactly once for each *Start* or *Extension* step in the formal calculus. Instead, a derivation is a proof if each literal in a leaf is included in a σ -complementary connection $\{L_1, L_2\}$.

Example 3. A proof of Pelletier’s problem 24a from Example 1 using the connection tableau representation is depicted in Figure 4. The rightmost literal in each proof step (number in parentheses) is annotated with the used clause number and (possibly) the term substitution that is applied to the clause. Proof step 1 is a *Start* step, step 7 is a *Reduction* step, all others are *Extension* steps.

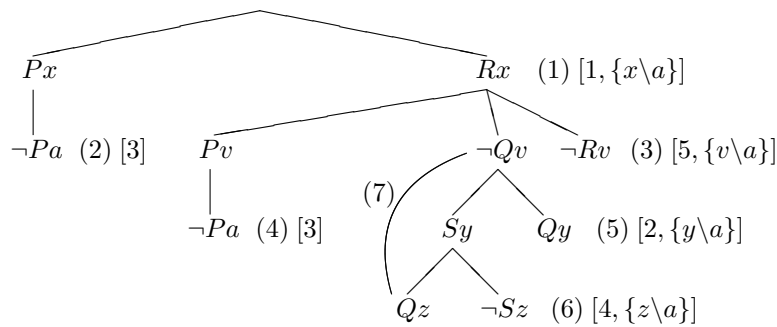


Figure 4: A proof for M_a using the connection tableau representation.

```

%-----
cnf(1, plain, [big_p(X), big_r(X)], clausify(pe124)).
cnf(2, plain, [big_s(X), big_q(X)], clausify(pe124_1)).
cnf(3, plain, [-(big_p(a))], clausify(pe124_3a)).
cnf(4, plain, [big_q(X), -(big_s(X))], clausify(pe124_4a)).
cnf(5, plain, [big_p(X), -(big_q(X)), -(big_r(X))], clausify(pe124_2)).

cnf('1', plain, [big_p(a), big_r(a)], start(1, bind([[X], [a]]))).
cnf('1.1', plain, [-(big_p(a))], extension(3)).
cnf('1.2', plain, [-(big_r(a)), big_p(a), -(big_q(a))], extension(5, bind([[X], [a]]))).
cnf('1.2.1', plain, [-(big_p(a))], extension(3)).
cnf('1.2.2', plain, [big_q(a), big_s(a)], extension(2, bind([[X], [a]]))).
cnf('1.2.2.1', plain, [-(big_s(a)), big_q(a)], extension(4, bind([[X], [a]]))).
cnf('1.2.2.1.1', plain, [-(big_q(a))], reduction('1.2')).
%-----

```

Figure 5: A proof of Pelletier’s problem 24a using the leanTPTP syntax as returned by leanCoP 2.2.

A connection proof using the leanTPTP syntax is a list of proof steps of the form

$$\text{cnf}(\textit{step_name}, \textit{plain}, \textit{clause}, \textit{rule_info}).$$

in which *step_name* is the name of the proof step, *clause* is the used (instantiated) clause C_2 , and *rule_info* has one of the following forms:

$$\begin{aligned} &\textit{start}(\textit{clause_number}, \textit{term_substitution}) \\ &\textit{reduction}(\textit{step_name}) \\ &\textit{extension}(\textit{clause_number}, \textit{term_substitution}) \end{aligned}$$

step_name is the name of the proof step and of the form ' $N_0.N_1\dots N_n$ ', in which the N_n -th “open” literal L_1 of its parent clause ' $N_0.N_1\dots N_{n-1}$ ' is used in a Reduction or Extension step; the name of the Start step ' N_0 ' is '1'. For example, '1.2.2' is the proof step from the second open literal of its parent clause in the proof step named '1.2'. *step_name* in Reduction steps refer to the proof step with the clause that contains the literal L_2 . *clause_number* is the number of the used *clause* in the matrix. *term_substitution* is of the form $\textit{bind}([\textit{vlist}, \textit{tlist}])$, in which *vlist* is a list of variables V_1, \dots, V_n and *tlist* is a list of terms t_1, \dots, t_n , such that $\sigma(V_i) = t_i$.

Example 4. *The proof of Pelletier’s problem 24a from Example 1 using the leanTPTP syntax is shown in Figure 5. It is produced by leanCoP 2.2 [14, 11] on the input file in Figure 3 (with simplified variable names and literals L_2 of Extension steps moved to the beginning of clauses).*

3.3. The CONNECT++ Syntax

CONNECT++ [6] is a C++ implementation of the connection calculus shown in Figure 1. The proof format of the CONNECT++ prover uses a stack to represent connection proofs. Each element on the stack corresponds to exactly one proof step in the (formal) connection calculus, and the stack is ordered to represent a depth-first search from the Start rule, exploring the left subtrees of Extensions first. Clauses, and literals within clauses, are numbered from 0. The first element of a proof is $\textit{start}(i)$, giving the index of the start clause used. Reductions are represented as $\textit{reduction}(p)$ where p denotes the index of the element in the path corresponding to L_2 .

```

matrix(0, [ -big_p(X), -big_r(X) ]).
matrix(1, [ -big_s(X), -big_q(X) ]).
matrix(2, [ big_p(a) ]).
matrix(3, [ -big_q(X), big_s(X) ]).
matrix(4, [ -big_p(X), big_q(X), big_r(X) ]).
proof_stack([
start(0),
left_branch(2, 0, 2), right_branch(2), left_branch(4, 2, 3),
left_branch(2, 0, 4), right_branch(4), left_branch(3, 0, 5),
left_branch(1, 0, 6), reduction(1), right_branch(6),
right_branch(5), right_branch(3)
]).

```

Figure 6: A proof of Pelletier’s problem 24a using the syntax as returned by CONNECT++.

Extensions are represented by `left_branch(i, j, d)` and `right_branch(d)` with i the index of C_2 , j the index of L_2 and d the proof tree depth.

Example 5. *The proof of Pelletier’s problem 24a from Example 1 using the CONNECT++ syntax is shown in Figure 6.⁴ In this proof, `start(0)` denotes that clause 0 is used by the Start rule, and `left_branch(2, 0, 2)` denotes that the first extension uses clause 2 and literal 0 within that clause, and is at depth 2 in the tree, and so on.*

Connect++ includes a proof checker. It is a short piece of Prolog code and uses Prolog’s resolution mechanism to build the substitution. However, it is straightforward to extend Connect++’s syntax to specify explicitly the substitution of the returned connection proof.

4. A Proposed Syntax for Connection Proofs

We propose the following syntax, which follows more closely the TPTP format described in Section 3.1 and generalizes the leanTPTP format presented in Section 3.2 with the following modifications:

- include information about the *parent* node, which allows the use of arbitrary proof step *names* in the first argument,
- add a list containing the elements of the active *path*,
- use the *inference* expression to specify clause number, substitutions and active path.

A connection proof using our proposed syntax is a list of proof steps of the form

`cnf(name, plain, formula, inference(rule_name, rule_info), parent)` .

in which *name* is the name of the proof step, *formula* is the used (instantiated) clause C_2 or a list containing L_2 of the Reduction step, *rule_name* is one of `start`, `reduction`, `extension` or any other used inference name (for example `lemma`), and *parent* is a list containing the name

⁴Note that the prover produces a slightly different proof of the problem by way of a different choice of C_2 in the third extension. Also, literals are negated throughout the matrix as CONNECT++ works in the negative (conjunctive normal form) representation, starting with the negation of the stated problem.


```

%-----
cnf(1, plain, [big_p(X), big_r(X)], clausify(pe124)).
cnf(2, plain, [big_s(X), big_q(X)], clausify(pe124_1)).
cnf(3, plain, [-(big_p(a))], clausify(pe124_3a)).
cnf(4, plain, [big_q(X), -(big_s(X))], clausify(pe124_4a)).
cnf(5, plain, [big_p(X), -(big_q(X)), -(big_r(X))], clausify(pe124_2)).

cnf(1, plain, [big_p(a), big_r(a)], inference(start, [1, bind([X], [a]), path([])], []).
cnf(2, plain, [-(big_p(a))], inference(extension, [3, path([big_p(a)])], [1])).
cnf(3, plain, [-(big_r(a)), big_p(a), -(big_q(a))],
    inference(extension, [5, bind([X], [a]), path([big_r(a)])], [1])).
cnf(4, plain, [-(big_p(a))], inference(extension, [3, path([big_r(a), big_p(a)])], [3])).
cnf(5, plain, [big_q(a), big_s(a)],
    inference(extension, [2, bind([X], [a]), path([big_r(a), -(big_q(a))])], [3])).
cnf(6, plain, [-(big_s(a)), big_q(a)],
    inference(extension, [4, bind([X], [a]), path([big_r(a), -(big_q(a)), big_s(a)])], [5])).
cnf(7, plain, [-(big_q(a))], inference(reduction, [3, path([big_r(a), -(big_q(a)), big_s(a)])], [6])).
%-----

```

Figure 7: A proof of Pelletier’s problem 24a using the proposed syntax.

of the previous (parent) proof step; finally, *rule_info* is a list of the form

$$[\textit{clause_name}, \textit{bind}([\textit{vlist}, \textit{tlist}]), \textit{path}(\textit{plist})]$$

in which *clause_name* is the name of the used clause, *vlist* is a list of variables V_1, \dots, V_n and *tlist* is a list of terms t_1, \dots, t_n , such that $\sigma(V_i) = t_i$; *plist* is the list of literals in the *active path*.

Example 6. *The proof of Pelletier’s problem 24a from Example 1 using the proposed syntax is shown in Figure 7.*

5. Conclusion

In this paper, we described the syntax of the connection proofs returned by two existing connection provers for classical first-order logic, and proposed a syntax for a proof format that fulfils the requirements listed in Section 2.1. As mentioned in the introduction, it should not be seen as a final specification, but as a foundation to initiate discussions within the community in order to finalize such a specification taking further proof formats [7, 4] into account.

A common standard for presentation of derivations and proofs will increase the interoperability between ATP systems, ATP tools, and application software. For example, Connect++ includes a tool to translate connection proofs into a LaTeX output of the formal calculus. A standardized syntax would make it possible to use this tool in combination with other connection provers. Such a standard would also allow the development of tools to translate connection proofs into sequent proofs [5], which are often used in interactive proof editors, such as Coq [1], NuPRL [3] or PVS [16]. To this end, a standardized syntax for sequent proofs would be desirable as well.

A possible extension of the current work includes the specification of a syntax to represent non-clausal connection proofs [12] or connection proofs in non-classical logics [10, 13]. These extensions should be kept in mind already when developing a format for clausal connection proofs in classical logics. For example, the connection calculi for intuitionistic and modal logic use an additional prefix substitution, which could be just added to each proof step.

References

- [1] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development - Coq'Art*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.
- [2] W. Bibel. *Automated Theorem Proving*. Vieweg and Sohn, 1987.
- [3] R. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.
- [4] M. Färber, C. Kaliszyk, and J. Urban. Machine Learning Guidance and Proof Certification for Connection Tableaux. arXiv:1805.03107v3 [cs.LO], 2018.
- [5] G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 36:176–210, 405–431, 1935.
- [6] S. B. Holden. Connect++: a fast, flexible and modifiable connection prover to support machine learning. In: J. Otten and W. Bibel, editors, *Proceedings of the Workshop on Automated Reasoning with Connection Calculi (AReCCa)*, 2023.
- [7] C. Kaliszyk, J. Urban, and J. Vyskočil. Certified Connection Tableaux Proofs for HOL Light and TPTP. In *Certified Programs and Proofs (CPP)*, pages 59–66, ACM, 2015.
- [8] R. Letz and G. Stenz. Model Elimination and Connection Tableau Procedures. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 2015–2114. Elsevier Science, 2001.
- [9] D.W. Loveland. Mechanical Theorem Proving by Model Elimination. *Journal of the ACM*, 15(2):236–251, 1968.
- [10] J. Otten. Clausal Connection-Based Theorem Proving in Intuitionistic First-Order Logic. In B. Beckert, editor, *TABLEAUX 2005*, LNAI, volume 3702, pages 245–261. Springer, 2005.
- [11] J. Otten. Restricting Backtracking in Connection Calculi. *AI Communications*, 23(2-3):159–182, 2010.
- [12] J. Otten. A Non-clausal Connection Calculus. In K. Brunnler, G. Metcalfe, editors, *TABLEAUX 2011*, LNAI, volume 6793, pages 226–241. Springer, 2011.
- [13] J. Otten. MleanCoP: A Connection Prover for First-Order Modal Logic. In S. Demri, D. Kapur, C. Weidenbach, editors, *IJCAR 2014*, LNAI, volume 8562, pages 269–276. Springer, 2014.
- [14] J. Otten and W. Bibel. leanCoP: Lean Connection-Based Theorem Proving. *Journal of Symbolic Computation*, 36(1-2):139–161, 2003.
- [15] J. Otten and G. Sutcliffe. Using the TPTP Language for Representing Derivations in Tableau and Connection Calculi. In B. Konev, R.A. Schmidt, S.]Schulz, editors, *PAAR 2010*, EPiC, volume 9, pages 95–105. EasyChair, 2010.
- [16] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification*, LNCS, volume 1102, pages 411–414. Springer, 1996.
- [17] F.J. Pelletier. Seventy-five Problems for Testing Automatic Theorem Provers. *Journal of Automated Reasoning*, 2(2):191–216, 1986.
- [18] G. Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 59(4):483–502, 2017.
- [19] G. Sutcliffe, S. Schulz, K. Claessen, and A. Van Gelder. Using the TPTP Language for Writing Derivations and Finite Interpretations. In U. Furbach and N. Shankar, editors, *IJCAR 2006*, LNAI, volume 4130, pages 67–81. Springer, 2006.