A Syntax for Connection Proofs

Jens Otten¹, Sean B. Holden²

¹Department of Informatics, University of Oslo, Norway
²Department 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 an improved syntax for connection proofs. Due to the general approach, the syntax can be extended to also represent connection proofs in some 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 and that can be verified by an external (independent) tool. Representing these proofs in a standardized format 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/TSTP syntax is well documented and specified [20]. 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 (formal) connection calculus consist of the axioms of the calculus, the leaves of a (TSTP) derivation in the resolution calculus consist 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).
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 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 syntaxes 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 given “proof” in at most polynomial time. This excludes, for example, “proofs” consisting only of a set of unsatisfiable clauses, for which finding a real proof is NP-complete.

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.

2.2. The Connection Calculus

Connection calculi, such as 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/list of clauses, where each clause is a set/list 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 calculi \([5]\) and (standard) 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 \( \sigma \) assigns terms to variables. A connection \( \{A_1, \neg A_2\} \) is \( \sigma \)-complementary iff \( \sigma(A_1) = \sigma(A_2) \).
Whenever a connection is identified. The connection calculus is sound and complete.

Axiom 1 labeled with the number of the used clause \( C \)

Translating these formulae into clausal form (shown in the third column) results in the matrix \( M_a \) and (the active) \( Path \) is a set of literals or \( \varepsilon \) and the term substitution \( \sigma \) is made by renaming all variables in \( C \). The \textit{rigid} term substitution \( \sigma \) is calculated by using one of the well-known \textit{term unification} algorithms whenever a connection is identified. The connection calculus is sound and complete [2].

**Definition 1 (Connection Proof).** A connection proof of \( M \) is a derivation of \( \varepsilon, M, \varepsilon \) with a term substitution \( \sigma \) in the connection calculus, in which all leaves are closed by axioms.

**Example 1.** A slightly simplified version\(^1\) of Pelletier’s problem 24 [17], called Pelletier 24a, consists of the following four axioms\(^2\) and one conjecture formulae.\(^3\)

\[
\begin{align*}
\exists x (P_x \land Rx) & \quad \text{Conjecture} & \{P_x, Rx\} & (1) \\
\neg (\exists x (Sx \land Qx)) & \quad \text{Axiom 1} & \{Sy, Qy\} & (2) \\
\exists x (P_x) & \quad \text{Axiom 3a} & \{\neg Pa\} & (3) \\
\forall x (Qx \Rightarrow Sx) & \quad \text{Axiom 4a} & \{Qv, \neg Sv\} & (4) \\
\forall (P_x \Rightarrow (Qx \lor Rx)) & \quad \text{Axiom 2} & \{Pz, \neg Qz, \neg Rz\} & (5)
\end{align*}
\]

Translating these formulae into clausal form (shown in the third column\(^4\)) results in the matrix \( M_a = \{\{P_x, Rx\}, \{Sy, Qy\}, \{\neg Pa\}, \{Qv, \neg Sv\}, \{Pz, \neg Qz, \neg Rz\}\}. The connection proof for \( M_a \) is shown below. Its graphical matrix representation is shown in Figure 2. The proof steps are labeled with the number of the used clause \( C \) and the term substitution applied to it.

\[
\begin{align*}
\{\}, M_a, \{Rx, \neg Qv, Sy\} & \quad A & \{Qz\}, M_a, \{Rx, \neg Qv, Sy\} & \quad R & \{\}, M_a, \{Rx, \neg Qv\} & \quad A & \{Sy\}, M_a, \{Rx, \neg Qv\} & \quad \text{E (4,} \{z\}a\}) & \quad \{\}, M_a, \{Rx\} & \quad A & \{Sy, Qy\} & \quad \text{E (2,} \{y\}a\}) \\
\{\}, M_a, \{Rx, Pv\} & \quad A & \{Pv, \neg Qv\}, M_a, \{Rx\} & \quad \{\neg Qv\}, M_a, \{Rx\} & \quad E (3) & \quad \{\}, M_a, \{\} & \quad A & \{Rx\}, M_a, \{\} & \quad \text{E (5,} \{v\}a\}) \\
\{\}, M_a, \{Px\} & \quad A & \{Pv, \neg Qv\}, M_a, \{Rx\} & \quad \{Rx\}, M_a, \{\} & \quad E (3) & \quad \{\}, M_a, \{\} & \quad A & \{Rx, \neg Qv\}, M_a, \{\} & \quad \text{E (1,} \{z\}a\}) \\
\{\}, M_a, \{\} & \quad \varepsilon, M_a, \varepsilon & \quad S (1, \{x\}a) & \quad \varepsilon, M_a, \varepsilon & \quad S (1, \{x\}a)
\end{align*}
\]

\(^1\)The original axioms 3 and 4 are \( \neg (\exists x P_x) \Rightarrow \exists y Qy \) and \( \forall x ((Qx \lor Rx) \Rightarrow Sx) \), respectively.

\(^2\)Note the difference between the axioms of the problem (the input formula) and the axiom of the calculus.

\(^3\)Axiom 2 is moved to the end of the set of axioms in order to comply with the proofs in Section 3.

\(^4\)A conjecture or axiom might result in more than one clause, which are marked with distinct numbers.
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/TSTP Syntax

The TPTP and TSTP languages are suitable for representing problems and derivations in first-order and higher-order logic [19, 20]. The top level building blocks are of the following form:

\[ \text{language}(\text{name}, \text{role}, \text{formula}, \text{source}, \text{useful_info}) \]

\text{language} is, for example, \texttt{fof} for first-order form or \texttt{cnf} for (conjunctive or disjunctive) clause normal form. Each formula has a unique \text{name}. \text{role} is a label such as \texttt{axiom} or \texttt{conjecture}. The \text{source} describes where the formula came from, for example an input file, and \text{useful_info} is a list of useful information. The last two fields are optional.

Example 2. Pelletier’s original problem 24 is included as problem SYN054+1 in the TPTP library. The representation of its simplified version 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 \text{source} has one of the forms

\[ \text{file}(/\text{file_name},/\text{file_info}) \]
\[ \text{inference}(/\text{inference_name},/\text{inference_info},/\text{parents}) \]

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

\begin{verbatim}
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) ) )).
\end{verbatim}

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.

<table>
<thead>
<tr>
<th>Start</th>
<th>Reduction</th>
<th>Extension</th>
</tr>
</thead>
</table>
| \[
C_2 = \{L_1, \ldots, L_n\}\text{ and } C_2 \text{ is copy of } C_1 \in M
\] | \[
L_2 \text{ is literal on path and } \{L_1, L_2\} \text{ is } \sigma\text{-complementary}
\] | \[
L_1 \text{ is leaf, } C_2 = \{L_1', \ldots, L_2', \ldots, L_n'\}\text{ and } C_2 \text{ is copy of } C_1 \in M, L_2 \in C_2,
\{L_1, L_2\} \text{ is } \sigma\text{-complementary}
\] |

3.2. The leanCoP Syntax

According to [20], 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/TSTP 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 \(\sigma\)-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.](image-url)
A connection proof using the leanTPTP syntax is a list of proof steps of the form
\[ \text{cnf}(\text{step}_\text{name}, \text{plain}, \text{clause}, \text{rule}_\text{info}) \]
in which \text{step}_\text{name} is the name of the proof step, \text{clause} is the used (instantiated) clause \( C_2 \), and \text{rule}_\text{info} has one of the following forms:

- \text{start}(\text{clause}_\text{number}, \text{term}_\text{substitution})
- \text{reduction}(\text{step}_\text{name})
- \text{extension}(\text{clause}_\text{number}, \text{term}_\text{substitution})

\text{step}_\text{name} is the name of the proof step and of the form ‘\text{N}_0.\text{N}_1......\text{N}_n’, in which the \text{N}_n-th “open” literal \( L_1 \) of its parent clause ‘\text{N}_0.\text{N}_1......\text{N}_{n-1}’ is used in a Reduction or Extension step; the name of the Start step ‘\text{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’. \text{step}_\text{name} in Reduction steps refer to the proof step with the clause that contains the literal \( L_2 \). \text{clause}_\text{number} is the number of the used \text{clause} in the matrix. \text{term}_\text{substitution} is of the form \text{bind}([vlist, tlist]), in which \text{vlist} is a list of variables \( V_1, \ldots, V_n \) and \text{tlist} is a list of terms \( t_1, \ldots, t_n \), 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 \text{start}(i), giving the index of the start clause used. Reductions are represented as \text{reduction}(p, q) where \( p \) denotes the index of the literal \( L_1 \) in \( C \cup \{L_1\} \) and \( q \)
denotes the index of the element in the path corresponding to \( L_2 \). Extensions are represented by \texttt{left\_branch}(p, i, j, d) and \texttt{right\_branch}(d) where \( p \) again denotes the index of the literal \( L_1 \) in \( C \cup \{L_1\} \), \( i \) is the index of \( C_2 \), \( j \) the index of \( L_2 \) and \( d \) the proof tree depth.

One aim of the \texttt{Connect++} syntax is to be quite explicit in stating how the rules of the connection calculus have been applied. This in turn stems from the idea that the simpler a proof checking program is, the more trustworthy it is likely to be, and this simplicity is supported by giving it as much information as possible. In the case of leanCoP, for example, a simplified version might be appropriate because \( L_1 \) is always the literal at index 0; however this might not be the case, for example, if a prover includes a heuristic for choosing an arbitrary literal with the aim of finding a proof more efficiently.

\textbf{Example 5.} The proof of Pelletier’s problem 24a from Example 1 using the \texttt{Connect++} syntax is shown in Figure 6.\footnote{Note 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 \texttt{Connect++} works in the negative (conjunctive normal form) representation, starting with the negation of the stated problem.} In this proof, \texttt{start(0)} denotes that clause 0 is used by the Start rule, and \texttt{left\_branch(0, 2, 0, 2)} denotes that the first Extension uses the first literal in \( C \cup \{L_1\} \), clause 2 and literal 0 within that clause, and is at depth 2 in the tree, and so on.

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

\section*{4. A Proposed Syntax for Connection Proofs}

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

\begin{itemize}
  \item include information about the \textit{parent} node, which allows the use of arbitrary proof step names in the first argument,
  \item use the \texttt{rule\_info} argument of the \texttt{inference} expression to specify clause name of \( C_1 \), position of literal \( L_1 \) in the parent clause, position of literal \( L_2 \) in \( C_2 \) (or in the active path), substitution \( \sigma \) applied to \( C_2 \) and the (instantiated) active \texttt{Path}.
\end{itemize}
A (clausal) connection proof using the proposed syntax is a list of proof steps of the form

\[ \text{cnf} (\text{name,plain,formula,inference(rule_name,rule_info,parent)}) \]

in which \text{name} is the name of the proof step, \text{formula} is the used (instantiated) clause \(C_2\) (in case of a Start or Extension step) or a list containing \(L_2\) (in case of a Reduction step),\(^6\) \text{rule\_name} is one of \text{start, reduction or extension}, and \text{parent} is a list containing the name of the previous (parent) proof step; furthermore, \text{rule\_info} is a list of the form

\[ [\text{clause\_name},\text{lit}(\text{lit1,lit2}),\text{bind}([\text{vlist},\text{tlist}]),\text{path}(\text{plist})] \]

in which \text{clause\_name} is the name of the (input) clause \(C_1\), \text{lit1} is the position of the literal \(L_1\) in the parent clause (in case of a Reduction or Extension step), \text{lit2} is the position of the literal \(L_2\) in the clause \(C_2\) (in case of an Extension step) or in the active path (in case of a Reduction step), \text{vlist} is a list of variables \(V_1,\ldots,V_n\) in \(C_2\) and \text{tlist} is a list of terms \(t_1,\ldots,t_n\), such that \(\sigma(V_i) = t_i\), \text{plist} is the list of literals in the active path.\(^7\) For the Start step, \text{parent} is empty, for Reduction steps, the argument \text{clause\_name} is omitted. The arguments \text{lit}, \text{bind} and \text{path} can be omitted.\(^8\)

\textbf{Example 6.} The proof of Pelletier’s problem 24a from Example 1 using the described syntax is shown in Figure 7.\(^9\) A simplified proof omitting redundant arguments is shown in Figure 8.
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 fulfills 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 [4, 7, 18, 21] into account.

Compared to the (previous) leanTPTP syntax, the proposed syntax follows more closely the TPTP/TSTP syntax, retaining the slightly simpler representation of proofs as connection tableau, while also including additional information, such as the active path. The latter information is useful when proof steps are to be carried out independently, e.g., in parallel, as all necessary information is included in the description of each proof step. Similar to the TPTP/TSTP format, the leanTPTP format and the Connect++ format, the proposed proof format uses Prolog terms to represent connection proofs. This allows the use of simple Prolog code to implement proof checkers as done, for example, in leanCoP and Connect++.

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 the rule_info argument. Furthermore, future work include the specification of proof formats for other popular proof calculi, such as sequent calculi or (standard) tableau calculi.
References


