Towards fully automated axiom extraction for finite-valued logics

João Marcos and Dalmo Mendonça

DIMAp / CCET, UFRN, Brazil

jmarcos@dimap.ufrn.br, dalmo3@gmail.com

Abstract

We implement an algorithm for extracting appropriate collections of classic-like sound and complete tableau rules for a large class of finite-valued logics. Its output consists of `Isabelle` theories.[1]

Keywords: many-valued logics, tableaux, automated theorem proving.

1 Introduction

This paper will report on the first developments towards the implementation of a fully automated program for the extraction of adequate proof-theoretical counterparts for sufficiently expressive logics characterized by way of a finite set of finite-valued truth-tables. The underlying algorithm was first described in [4]. Surveys on tableaux for many-valued logics can be found in [7, 1]. Our implementation has been performed in `ML`, and its application gives rise to an `Isabelle` theory (check [8]) formalizing a given finite-valued logic in terms of two-signed tableau rules.

The survey paper [7] points at a few very good theoretical motivations for studying tableaux for many-valued logics, among them:

- tableau systems are a particularly well-suited starting point for the development of computational insights into many-valued logics;

- a close interplay between model-theoretic and proof-theoretic tools is necessary and fruitful during the development of proof procedures for non-classical logics.

Section 2, right below, recalls the relevant definitions and some general results concerning many-valued logics as well as their homologous presentation in terms of bivalent semantics described by clauses of a certain format we call 'gentzenian'. An algorithm for endowing any sufficiently expressive finite-valued logic with an adequate bivalent semantics is exhibited and illustrated for the case of Ł$_3$, the well-known 3-valued logic of Łukasiewicz.

---

[1] A development snapshot of the code may be found at `http://tinyurl.com/5cakro`.

The main concepts concerning tableau systems in general and the particular results that allow one to transform any computable gentzenian semantics into a corresponding collection of tableau rules are illustrated in section 3, again for the case of L_3.

Section 4 discusses our current implementation, carefully explaining its expected inputs and outputs, and yet again illustrates its functioning for the case of L_3. Advantages and shortcomings of our program, in its present state of completion, as well as conclusions and some directions for future developments are mentioned in section 5.

2 Many-valued logics

Given a denumerable set At of *atoms* and a finite family $\mathsf{Cct} = \{©_j^i\}_{j \in J}$ of *connectives*, where $\mathsf{arity}(©_j^i) = i$, let $\mathcal{S}$ denote the term algebra freely generated by Cct over At. Here, a *semantics* Sem for the algebra $\mathcal{S}$ will be given by any family of mappings $\{\S_k^{\mathcal{V}}\}_{k \in K}$ where $\mathsf{dom}(\S_k^{\mathcal{V}}) = \mathcal{S}$ and $\mathsf{codom}(\S_k^{\mathcal{V}}) = \mathcal{V}_k$, and where each collection of *truth-values* $\mathcal{V}_k$ is partitioned into sets of designated values, $\mathcal{D}_k$, and undesignated ones, $\mathcal{U}_k$. The mappings $\S_k^{\mathcal{V}}$ themselves may be called *(κ-valued) valuations*, where $\kappa = \mathsf{Card}(\mathcal{V}_k)$. A *bivalent semantics* is any semantics where $\mathcal{D}_k$ and $\mathcal{U}_k$ are singleton sets, for every $k \in K$. For bivalent semantics, valuations are often called *bivaluations*. The canonical notion of (single-conclusion) *entailment* $\models_{\mathsf{Sem}} \subseteq \mathsf{Pow}(\mathcal{S}) \times \mathcal{S}$ induced by a semantics Sem is defined by setting $\Gamma \models_{\mathsf{Sem}} \varphi$ iff $\S_k^{\mathcal{V}}(\varphi) \in \mathcal{D}_k$ whenever $\S_k^{\mathcal{V}}(\Gamma) \subseteq \mathcal{D}_k$, for every $\S_k^{\mathcal{V}} \in \mathsf{Sem}$. The pair $\langle \mathcal{S}, \models_{\mathsf{Sem}} \rangle$ may then be called a *generic κ-valued logic*, where $\kappa = \mathsf{Max}_{k \in K}(\mathsf{Card}(\mathcal{V}_k))$.

If one now fixes the sets of truth-values $\mathcal{V}$, $\mathcal{D}$ and $\mathcal{U}$, and fixes, for each connective $©_j^i$ an interpretation $\widehat{©}_j^i : \mathcal{V}^i \longrightarrow \mathcal{V}$, one may immediately build from that an associated algebra of truth-values $\mathcal{TV} = \langle \mathcal{V}, \mathcal{D}, \{\widehat{©}_j^i\}_{j \in J} \rangle$ (in the present paper, whenever there is no risk of confusion, we shall not differentiate notationally between a connective symbol $©$ and its operational interpretation $\widehat{©}$). A *truth-functional* semantics is then defined by the collection of all homomorphisms of $\mathcal{S}$ into $\mathcal{TV}$. In the present paper, the shorter expression *κ-valued logic* (or, in general, *finite-valued logic*) will be used to qualify any generic κ-valued truth-functional logic, for some finite κ, where κ is the minimal value for which the mentioned logic can be given a truth-functional semantics characterizing the same associated notion of entailment.

It is interesting to observe that the canonical notion of entailment of any given semantics, and in particular of any given truth-functional semantics, may be emulated by a bivalent semantics. Indeed, let $\mathcal{V}^2 = \{F, T\}$ and $\mathcal{D}^2 = \{T\}$, and consider the 'binary print' of the algebraic truth-values produced by the total mapping $t : \mathcal{V} \longrightarrow \mathcal{V}^2$, defined by $t(v) = T$ iff $v \in \mathcal{D}$. For any κ-valued valuation $\S^{\mathcal{V}}$ of a given semantics Sem, consider now the characteristic total function $b_\S = t \circ \S^{\mathcal{V}}$. Now, collect all such bivaluations $b_\S$'s into a new semantics $\mathsf{Sem}(2)$, and note that $\Gamma \models_{\mathsf{Sem}(2)} \varphi$ iff $\Gamma \models_{\mathsf{Sem}} \varphi$. As a matter of fact, the standard 2-valued notion of inference of Classical Logic is characterized indeed by

a finite-valued semantics that is simultaneously bivalent and truth-functional. In general, nonetheless, if a logic is κ-valued, for $\kappa > 2$, a bivalent characterization of it will explore the trade-off between, on the one hand, the 'algebraic perspective' of many-valuedness, with its many 'algebraic truth-values' and its semantic characterization in terms of a set of homomorphisms, and, on the other hand, the classic-inclined 'logical perspective', with its emphasis on characterizations based on two 'logical values' (for more detailed discussions of this issue, check [4, 11]). Our interest in this paper is to probe some of the practical advantages of the bivalent classic-like perspective as applied to the wider domain of finite-valued truth-functional logics.

Our running example in the present paper will involve Łukasiewicz's well-known 3-valued logic $Ł_3$, characterized by the algebra of truth-values $\langle \{1, \frac{1}{2}, 0\}, \{1\}, \{\neg, \rightarrow, \vee, \wedge\} \rangle$, where the interpretation of the unary negation connective $\neg$ sets $\neg v_1 = 1 - v_1$ and the interpretation of the binary implication connective $\rightarrow$ sets $(v_1 \rightarrow v_2) = \mathsf{Min}(1, 1 - v_1 + v_2)$. The binary symbols $\vee$ and $\wedge$ can be introduced as primitive if we interpret them by setting $(v_1 \vee v_2) = \mathsf{Max}(v_1, v_2)$ and $(v_1 \wedge v_2) = \mathsf{Min}(v_1, v_2)$, but they can also, more simply, be introduced by definition just like classical disjunction and conjunction, setting $\alpha \vee \beta \stackrel{\text{def}}{=} (\alpha \rightarrow \beta) \rightarrow \beta$ and $\alpha \wedge \beta \stackrel{\text{def}}{=} \neg(\neg\alpha \vee \neg\beta)$. The binary prints of an arbitrary atom of $Ł_3$ and of its negation are illustrated in the table below.

$$
\begin{array}{|c|c||c|c|}
\hline
v & t(v) & \neg v & t(\neg v) \\
\hline
0 & F & 1 & T \\
\hline
\frac{1}{2} & F & \frac{1}{2} & F \\
\hline
1 & T & 0 & F \\
\hline
\end{array}
\tag{1}
$$

Given some finite-valued logic $\mathcal{L}$ based on a set of truth-values $\mathcal{V}$, we say that $\mathcal{L}$ is *functionally complete* over $\mathcal{V}$ if any κ-valued n-ary operation, for $\kappa = \mathsf{Card}(\mathcal{V})$, is definable with the help of a suitable combination of its primitive operators $\{\widehat{\mathbb{C}}_j^i\}_{j \in J}$. When $\mathcal{L}$ is not functionally complete from the start, we may consider $\mathcal{L}^{\mathbf{fc}}$ as any functionally complete κ-valued conservative extension of $\mathcal{L}$. Given truth-values $v_1, v_2 \in \mathcal{V}$, we say that they are *separated*, and we write $v_1 \sharp v_2$, in case v_1 and v_2 belong to different classes of truth-values, that is, in case either $v_1 \in \mathcal{D}$ and $v_2 \in \mathcal{U}$, or $v_1 \in \mathcal{U}$ and $v_2 \in \mathcal{D}$. Given a unary, primitive or defined, connective $\circledS$ of a given truth-functional logic, with interpretation $\widehat{\circledS}$, we say that $\circledS$ *separates* v_1 and v_2 in case $\widehat{\circledS}(v_1) \sharp \widehat{\circledS}(v_2)$. Obviously, for any pair of truth-values in $\mathcal{V}$ it is possible to define in the term algebra of $\mathcal{L}^{\mathbf{fc}}$ an appropriate *separating connective* $\circledS$. When that separation can be done exclusively with the help of the original language of $\mathcal{L}$, we say that $\mathcal{V}$ is *effectively separable*, and the logic $\mathcal{L}$, in that case, will be considered to be *sufficiently expressive* for our purposes. It should be noticed that the vast majority of the most well-known finite-valued logics enjoy this expressivity property.

Notice in particular, from Table (1) above, how the negation connective of $Ł_3$ separates the two undesignated truth-values, $\frac{1}{2}$ and 0. Based on this table, one may in fact easily provide a unique identification to each of the 3 initial algebraic

truth-values of Ł$_3$, by way of the 3 following statements:

$$v = 1 \qquad \text{iff} \qquad t(v) = T \tag{I}$$
$$v = \tfrac{1}{2} \qquad \text{iff} \qquad t(v) = F \text{ and } t(\neg v) = F$$
$$v = 0 \qquad \text{iff} \qquad t(v) = F \text{ and } t(\neg v) = T$$

One can also use this separating connective $Ⓢ = \lambda u.\neg u$ in order to provide a bivalent description of each of the operators of the language. Consider for instance the cases of $ⓒ_1 = \lambda vw.(v \to w)$ and $ⓒ_2 = \lambda vw.\neg(v \to w)$ (that is, $ⓒ_2$ is $Ⓢⓒ_1$):

$ⓒ_1$	0	$\frac{1}{2}$	1
0	1	1	1
$\frac{1}{2}$	$\frac{1}{2}$	1	1
1	0	$\frac{1}{2}$	1

$ⓒ_2$	0	$\frac{1}{2}$	1
0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0
1	1	$\frac{1}{2}$	0

$$\tag{2}$$

From table $ⓒ_2$ it is clear for instance that:

$$\S(\neg(\alpha \to \beta)) = 1 \qquad \text{iff} \qquad \S(\alpha) = 1 \text{ and } \S(\beta) = 0 \tag{II}$$

Let's write $T{:}\varphi$ and $F{:}\varphi$, respectively, as abbreviations for $t(\S(\varphi)) = T$ and $t(\S(\varphi)) = F$. Then, the statement (II) may be described in bivalent form, with the help of (I), by writing:

$$T{:}\neg(\alpha \to \beta) \qquad \text{iff} \qquad T{:}\alpha \text{ and } (F{:}\beta \text{ and } T{:}\neg\beta) \tag{III}$$

In [4] an algorithm that constructively specifies a bivalent semantics for any sufficiently expressive finite-valued logic was proposed. The output of the algorithm is a computable class of clauses governing the behavior of all the bivaluations that together will canonically define an entailment that coincides with the original entailment defined with the help of the algebra of truth-values $\mathcal{TV}$ and the class of all the corresponding finite-valued homomorphisms of $\mathcal{S}$ into $\mathcal{TV}$. Moreover, all those clauses are in a specific format we call *gentzenian*, namely, they are conditional expressions (E) of the form $\Phi \Rightarrow \Psi$ where the symbol $\Rightarrow$ represents a meta-linguistic implication, and both Φ and Ψ are meta-formulas of the form $\top$ (top), $\bot$ (bottom) or a clause (G) of the form $\bigvee_{1 \leq i \leq m} \bigwedge_{1 \leq j \leq n_m} A(i,j,w)$, where each $A(i,j,w)$ has the form $b(\varphi_i^j) = w_i^j$, for some given $\varphi_i^j \in \mathcal{S}$ and $w_i^j \in \{F,T\}$. (Recall that we use $b : \mathcal{S} \to \mathcal{V}^2$ for bivaluations.) If we let $|$ represent disjunction and $\&$ represent conjunction in the meta-language, any clause (G) will have thus the extended format:

$$b(\varphi_1^1) = w_1^1 \& \ldots \& b(\varphi_1^{n_1}) = w_1^{n_1} \mid \ldots \mid b(\varphi_m^1) = w_m^1 \& \ldots \& b(\varphi_m^{n_m}) = w_m^{n_m}.$$

The meta-logic governing such meta-linguistic expressions is FOL, First-Order Classical Logic. Accordingly, an expression of the form $A_1|A_2 \Rightarrow A_3$ will be equivalent to $(A_1 \Rightarrow A_3)\&(A_2 \Rightarrow A_3)$, and an expression of the form $A_1\&A_2 \Rightarrow A_3$ will be equivalent to $A_1 \Rightarrow A_3|{\sim}A_2$, where the meta-linguistic negation $\sim$ is

such that $\sim(b(\varphi) = T)$ denotes $b(\varphi) = F$, and $\sim(b(\varphi) = F)$ denotes $b(\varphi) = T$. We may also write $\Phi \Leftrightarrow \Psi$ as an abbreviation for $(\Phi \Rightarrow \Psi)\&(\Psi \Rightarrow \Phi)$.

With a slight notational change and using FOL, one can now see (III) as a description done in an abbreviated gentzenian format:

$$T{:}\neg(\alpha \to \beta) \quad \Leftrightarrow \quad T{:}\alpha \ \& \ F{:}\beta \ \& \ T{:}\neg\beta \qquad\qquad\qquad (IV)$$

Following the above line of reasoning, and considering now table $\copyright_1$ instead of $\copyright_2$, it is also correct to write, for instance, the clause:

$$
\begin{aligned}
F{:}(\alpha \to \beta) \quad \Leftrightarrow \quad & T{:}\alpha \ \& \ F{:}\beta \ \& \ F{:}\neg\beta \ | \qquad\qquad (V)\\
& T{:}\alpha \ \& \ F{:}\beta \ \& \ T{:}\neg\beta \ |\\
& F{:}\alpha \ \& \ F{:}\neg\alpha \ \& \ F{:}\beta \ \& \ T{:}\neg\beta
\end{aligned}
$$

According to the reductive algorithm described in [4], a sound and complete bivalent version of any sufficiently expressive finite-valued logic $\mathcal{L}$ is obtained if one performs the following two steps:

[step 1] Iterate the above illustrated procedure in order to obtain clauses describing exactly in which situations one may assert the sentences $T{:}\copyright(\alpha_1, \ldots, \alpha_n)$, $F{:}\copyright(\alpha_1, \ldots, \alpha_n)$, $T{:}\circledS\copyright(\alpha_1, \ldots, \alpha_n)$ and $F{:}\circledS\copyright(\alpha_1, \ldots, \alpha_n)$, for each primitive n-ary $\copyright \in \mathsf{Cct}$ and each one of the separating connectives $\circledS$ of $\mathcal{L}$.

[step 2] Add to all those clauses the following extra axioms governing the behavior of the admissible collection of bivaluations:

    (C1) $\top \ \Rightarrow \ T{:}\alpha \ | \ F{:}\alpha$

    (C2) $T{:}\alpha \ \& \ F{:}\alpha \ \Rightarrow \ \bot$

    (C3) $T{:}\alpha \ \Rightarrow \ \bigvee_{d \in \mathcal{D}} \bigwedge_{1 \le m < n \le \mathsf{Card}(\mathcal{D})} w_{mn}^d{:}\circledS_{mn}(\alpha)$

    (C4) $F{:}\alpha \ \Rightarrow \ \bigvee_{u \in \mathcal{U}} \bigwedge_{1 \le m < n \le \mathsf{Card}(\mathcal{U})} w_{mn}^u{:}\circledS_{mn}(\alpha)$

for every $\alpha \in \mathcal{S}$, where $\circledS_{mn}$ is the unary (primitive or defined) connective that we use to separate the truth-values m and n, and $w_{mn}^v = t(\circledS_{mn}(v))$.

Notice that (C3) and (C4) have the role of recording the relevant information on binary prints, as it can be found, in the case of L_3, in Table 1.

3   Tableaux

Generic tableau systems for finite-valued logics are known at least since [5]. In the corresponding tableaux, however, formulas may receive as many labels as the number of truth-values in $\mathcal{V}$, and that somewhat obstructs the task of comparing for instance the associated notions of proof and of consequence relation to the corresponding classical notions. However, as it will be shown, with the help of the bivalent semantics illustrated in the previous section it is now straightforward to produce sound and complete collections of classic-like two-signed tableau rules (i.e., each formula appears with exactly one of *two* labels at the head of each rule).

The basic idea, explained in [4], is to dispose the gentzenian clauses governing the admissible bivaluations in an appropriate way. For that matter, clauses such as (IV) and (V) can be rendered, respectively, into the following tableau rules:

$(\text{IV})^{tab}$ $T{:}\neg(\alpha \rightarrow \beta)$ $F{:}(\alpha \rightarrow \beta)$ $(\text{V})^{tab}$

$$\begin{array}{c}
T{:}\neg(\alpha \rightarrow \beta) \\
| \\
T{:}\alpha \\
F{:}\beta \\
T{:}\neg\beta
\end{array}
\qquad\qquad
\begin{array}{ccc}
 & F{:}(\alpha \rightarrow \beta) & \\
T{:}\alpha & T{:}\alpha & F{:}\alpha \\
F{:}\beta & F{:}\beta & F{:}\neg\alpha \\
F{:}\neg\beta & T{:}\neg\beta & F{:}\beta \\
 & & T{:}\neg\beta
\end{array}$$

Each tableau branch can be seen as a non-empty sequence of signed formulas, $\Gamma, L{:}\varphi, \Delta$, where $L \in \{F, T\}$, $\varphi \in \mathcal{S}$ and Γ and Δ are sequences. To be sure, a rule such as $(\text{V})^{tab}$ will be read thus as an instruction allowing one to transform a sequence $\Gamma, F{:}(\alpha \rightarrow \beta), \Delta$ into the three following new sequences:

[1] $\Gamma, T{:}\alpha, F{:}\beta, F{:}\neg\beta, \Delta$

[2] $\Gamma, T{:}\alpha, F{:}\beta, T{:}\neg\beta, \Delta$

[3] $\Gamma, T{:}\alpha, F{:}\neg\alpha, F{:}\beta, F{:}\neg\beta, \Delta$

If one recalls the first step of the algorithm described in the last section, there will be tableau rules with heads $L{:}\text{ⓒ}\varphi$ and $L{:}\text{ⓢ}\text{ⓒ}\varphi$ for each sign $L \in \{F, T\}$, each primitive connective ⓒ and each separating connective ⓢ. Rules $(\text{IV})^{tab}$ and $(\text{V})^{tab}$, above, are indeed examples of such tableau rules. To obtain soundness and completeness with respect to the original finite-valued semantics, in general, besides the rules produced in the first step one must also take into consideration the rules corresponding to axioms (C1)–(C4), mentioned in the second step. It is interesting to notice that in most practical cases, however, axioms (C3) and (C4) can often be directly proven from the remaining ones. Moreover, axiom (C2) expresses just the usual closure condition on tableau branches. On the other hand, axiom (C1) gives rise in general to the following *dual-cut* rule, for arbitrary α:

$$T : \alpha \qquad F : \alpha$$

All further definitions and concepts concerning the setup and construction of signed tableaux are standard (check [9]). In particular, a *branch* (sequence of signed formulas) is said to be *closed* if there is a formula that occurs in it both with the sign T and with the sign F, and a *closed tableau* for a given sequence of signed formulas is produced by a finite set of transformations allowed by the corresponding tableau rules to such an effect that at some point all branches that originate from the original branch are closed.

One could be worried, and with good reason, that the unrestrained use of the dual-cut rule might potentially make the corresponding tableaux non-analytic. We will discuss that issue in the conclusion. The tableau rules originated from

6

the above procedure can naturally be used in order to prove theorems, check conjectures and suggest counter-models, but also, in the meta-theory, to formulate and prove derived rules that can be used to simplify the original presentation of the logic as originated by our algorithm. So, for instance, the above illustrated complex three-branching rule for $F : (\alpha \to \beta)$ can eventually be simplified into any one of the following equivalent two-branching rules:

$$(\mathrm{V}^*)^{tab} \qquad \begin{array}{c} F : (\alpha \to \beta) \\ \diagdown \\ \begin{array}{cc} T : \alpha & F : \alpha \\ F : \beta & F : \neg\alpha \\ & F : \beta \\ & T : \neg\beta \end{array} \end{array} \qquad\qquad \begin{array}{c} F : (\alpha \to \beta) \\ \diagdown \\ \begin{array}{cc} T : \alpha & F : \neg\alpha \\ F : \beta & T : \neg\beta \end{array} \end{array} \qquad (\mathrm{V}^{**})^{tab}$$

Tableau systems for the logic L_3, in particular, are known indeed at least since [10]. The latter have not been generated algorithmically, though, through an automated generic procedure such as the one we illustrate here.

4 Implementation, and applications

We used the functional programming language `ML` to automate the axiom extraction process. `ML` provides one, among other advantages, with an elegant and suggestive syntax, a compile-time type checking that is both modular and reliable, as well as a type inference mechanism that greatly helps both in preventing program errors and in the task of formal verification of correctness of the implemented algorithms.

The relevant inputs of our program include the detailed definition of a finite-valued logic, together with an appropriate set of separating connectives for that logic. Here's an example of an input for the logic L_3, presented above, where the functions `CSym`, `CAri` and `CPre` take a connective and return its symbol (syntactic sugar), arity and precedence/associativity rules, respectively. The primitive connectives are defined by their truth-tables, listed in `CTabs`. A truth-table of an n-ary connective $\copyright$ is represented as a list of all pairs ([$x_1, \ldots, x_n$], y) such that $\copyright(x_1, \ldots, x_n) = y$. Derived connectives should be defined by abbreviation in terms of the primitive ones, and these definitions are given by the function `CDef`.

```
(* PROGRAM INPUT, EXAMPLE OF L3 *)
  structure L3 : LOGIC =
struct
   val theoryName  = "TL3";
   val Values      = ["0", "1/2", "1"];
   val Designated  = ["1"];
   val Connectives = ["Neg", "Imp", "Disj", "Conj"];
   val Primitives  = ["Neg", "Imp"];
   val SeparatingD = [];
   val SeparatingU = ["Neg"];
```

```
      fun CSym "Neg"  = "~"
        | CSym "Imp"  = "-->"
        | CSym "Disj" = "|"
        | CSym "Conj" = "&";

      fun CAri "Neg"  = 1
        | CAri "Imp"  = 2
        | CAri "Disj" = 2
        | CAri "Conj" = 2;

      val CTabs = ref [
            ("Neg", [ (["0"],    "1"),
                      (["1/2"], "1/2"),
                      (["1"],    "0")] ) ,
            ("Imp", [ (["0", "0"],     "1"),
                      (["0", "1/2"],   "1"),
                      (["0", "1"],     "1"),
                      (["1/2", "0"],   "1/2"),
                      (["1/2", "1/2"], "1"),
                      (["1/2", "1"],   "1"),
                      (["1", "0"],     "0"),
                      (["1", "1/2"],   "1/2"),
                      (["1", "1"],     "1")] )
            ];

      fun CDef "Disj" = ( "A0 | A1", "(A0 --> A1) --> A1" )
        | CDef "Conj" = ( "A0 & A1", "~(~A0 | ~A1)" );

      fun CPre "Neg"  = "[40] 40"
        | CPre "Imp"  = "[34,35] 35"
        | CPre "Disj" = "[24,25] 25"
        | CPre "Conj" = "[29,30] 30";
end;
```

For every symbol from `Connectives` that may appear in both the list of `Primitives`
and the list of rewrite rules `CDef`, our program calculates its corresponding truth-
table in terms of those that can be found in `CTab`, and adds it to the input of
the algorithm for extracting tableau rules. (A future version of the program
shall handle truth-tables defined intensionally, by way of a lambda-calculus-like
expression.)

To perform the extraction, our program first generates a list of heads for
all the necessary rules, according to the algorithm explained and illustrated in
sections 2 and 3.

```
val rulesList = [T "~(A0)",    T "A0 --> A1",
                 T "~(~(A0))", T "~(A0 --> A1)",
                 F "~(A0)",    F "A0 --> A1",
                 F "~(~(A0))", F "~(A0 --> A1)" ]
```

Obviously, in general, if there are c primitive connectives and s separating con-
nectives, there will be exactly $2 \times c \times s$ rule heads in `rulesList`.

Next, each connective's truth-table is converted into a table where each
value is exchanged by its binary print. The binary print of a value is calculated

based on the separating connectives given as input (`SeparatingD` for designated values and `SeparatingU` for undesignated values). The program represents such tables as lists of pairs (`input, output`). For the case of `Neg`, the sole separating connective of our version of Ł$_3$, the table contains the following information:

```
          (* A0 *)                 (* ~(A0) *)
 (* 0    *)  ( [F "A0", T "~(A0)"],   [T "~(A0)"] )
 (* 1/2 *)  ( [F "A0", F "~(A0)"],   [F "~(A0)", F "~(~(A0))"] )
 (* 1    *)  ( [T "A0"],              [F "~(A0)", T "~(~(A0))"] )
```

Now, for each expression in `rulesList`, a search through all tables generated in the last step is done, and all clauses in which the given formula appears on the right-hand side (as output) are returned. The left-hand side (input) of these clauses are the branches of the desired tableau rule. As an example, two of the clauses involving implication, and corresponding to the tableau rules $(IV)^{tab}$ and $(V)^{tab}$ from the last section, are calculated and recorded by the program as follows:

```
(* IV *) ([ [T "A0",F "A1",T "~(A1)"] ],          T "~(A0 --> A1)")
(* V  *) ([ [F "A0",F "~(A0)",F "A1",T "~(A1)"],
           [T"A0",F "A1",T "~(A1)"],
           [T"A0",F "A1",F "~(A1)"] ] ,          F "A0 --> A1")
```

The next, and final, steps include calculating axioms (C3) and (C4), and printing all definitions, syntactical details and rules into a theory file, ready to be used by `Isabelle`.

`Isabelle`, also written in `ML`, is a generic theorem-proving environment based on a higher-order meta-logic in which it is quite simple to create formal theories with rules and axioms for various kinds of deductive formalisms, and equally easy to define tacticals and to prove theorems about the underlying formal systems.

Here's an illustration of how we use `Isabelle`'s syntax for representing tableaux, extending the theory `Sequents.thy` that comes with `Isabelle`'s default library, and taking as example the file generated by our program as output for the logic Ł$_3$:

```
theory TL3

imports Sequents
begin
typedecl a

consts
  Trueprop :: "(seq'=>seq') => prop"
  TR       :: "a => o"           ("T:_" [20] 20)
  FR       :: "a => o"           ("F:_" [20] 20)
  Neg      :: "a => a"           ("~ _" [40] 40)
  Imp      :: "[a,a] => a"       ("_-->_" [24,25] 25)
syntax "@Trueprop"      ::  "(seq) => prop"  ("[_]" 5)
```

9

```
ML
{*
fun seqtab_tr  c [s] =  Const(c,dummyT) $ seq_tr s;
fun seqtab_tr' c [s] =  Const(c,dummyT) $ seq_tr' s;
*}

parse_translation {* [("@Trueprop", seqtab_tr "Trueprop")] *}
print_translation {* [("Trueprop", seqtab_tr' "@Trueprop")] *}

local
axioms

axC1:  "[| [ $H, T:A ] ; [ $H, F:A ] |] ==> [$H]"

axC21: "[ $H, T:A, $E, F:A, $G]"
axC22: "[ $H, F:A, $E, T:A, $G]"

axC3:  "[| [$H, T:A, $G ] |]
            ==> [ $H, T:A, $G ]"
axC4:  "[| [ $H, F:A, T:~(A), $G ] ;
            [ $H, F:A, F:~(A), $G ] |]
            ==> [ $H, F:A, $G ]"

ax0:   "[| [ $H, F:A0, T:~(A0), $G ] |]
            ==> [ $H, T:~(A0), $G ]"

ax1:   "[| [ $H, F:A0, T:~(A0), F:A1, T:~(A1), $G ] ;
            [ $H, T:A0, T:A1, $G ] ;
            [ $H, F:A0, F:~(A0), T:A1, $G ] ;
            [ $H, F:A0, F:~(A0), F:A1, F:~(A1), $G ] ;
            [ $H, F:A0, T:~(A0), T:A1, $G ] ;
            [ $H, F:A0, T:~(A0), F:A1, F:~(A1), $G ] |]
            ==> [ $H, T:A0 --> A1, $G ]"

ax2:   "[| [ $H, T:A0, $G ] |]
            ==> [ $H, T:~(~(A0)), $G ]"

ax3:   "[| [ $H, T:A0, F:A1, T:~(A1), $G ] |]
            ==> [ $H, T:~(A0 --> A1), $G ]"

ax4:   "[| [ $H, F:A0, F:~(A0), $G ] ;
            [ $H, T:A0, $G ] |]
            ==> [ $H, F:~(A0), $G ]"

ax5:   "[| [ $H, F:A0, F:~(A0), F:A1, T:~(A1), $G ] ;
            [ $H, T:A0, F:A1, F:~(A1), $G ] ;
            [ $H, T:A0, F:A1, T:~(A1), $G ] |]
            ==> [ $H, F:A0 --> A1, $G ]"

ax6:   "[| [ $H, F:A0, F:~(A0), $G ] |]
            ==> [ $H, F:~(~(A0)), $G ]"

ax7:   "[| [ $H, F:A0, F:~(A0), F:A1, T:~(A1), $G ] ;
            [ $H, T:A0, F:A1, F:~(A1), $G ] |]
            ==> [ $H, F:~(A0 --> A1), $G ]"

ML {* use_legacy_bindings (the_context ()) *}
```

```
(* Abbreviations *)
Conj_def:   "A0 & A1 == ~(~A0 | ~A1)"
Disj_def:   "A0 | A1 == (A0 --> A1) --> A1"

(* Structural rules *)
thin:       "[$H, $E, $G] ==> [$H, $A, $E, $B, $G]"
exch:       "[$H, $A, $E, $B, $G] ==> [$H, $B, $E, $A, $G]"

end
```

In this theory, `consts` lists the formula constructors. `TR ::  "a => o"` means that the constructor `TR` takes a formula (typed `a`) and returns a labeled formula (typed `o`). The addition of the structural rules, unusual for tableau systems, is motivated by the fact that our formalism for tableaux was based on `Isabelle`'s `Sequents.thy`, but also because we want to be able to prove not just *theorems* but also *derived rules*. A detailed example of such proofs will be presented below.

In the generated axioms corresponding to the tableau rules, `T:X` and `F:X` are (labeled) formulas, `$X`, is any sequence of such formulas, or contexts, each sequence between square brackets represents a tableau branch, and a collection of branches is delimited by `[|` and `|]`. The symbol `==>` denotes `Isabelle`'s meta-implication, that may be used to write down object-language rules, and `==` denotes `Isabelle`'s meta-equality, that may be used for stating rewrite rules concerning the derived operators of our logics. In `Isabelle`, the application of a rule means that it's possible to achieve the goal (sequence on the right of the meta-implication) once it's possible to prove the hypotheses (sequences on the left of the meta-implication), which constitute the collection of new subgoals that take the place of the original goal after the rule is applied. The dual-cut rule corresponds to axiom `axC1`, and the closure rule for a branch of the tableau corresponds to the axioms `axC21` and `axC22`. We retain both the latter rules as primitive for reasons of efficiency, but clearly one is derivable from the other with the use of of `exch`. Notice in particular how, in effect, the tableau *rules* produced for the original finite-valued logic provided as input correspond to *axioms* in the higher-order language of `Isabelle`.

The axiom set generated by the generic algorithm that we implemented isn't necessarily the most smart or efficient one. Axiom `axC3`, for instance, is clearly ineffectual in the above theory, having the same sequences at both sides of the meta-implication. Moreover, it may also happen that the formula over which the rule is applied also appears in some of the resulting branches, yet one is likely to try to control that phenomenon when aiming at defining tacticals for automated theorem proving.

Note that `ax5` corresponds to rule $(V)^{tab}$ from the last section. The simpler rule $(V^{**})^{tab}$, mentioned in the same section, can obviously be written in `Isabelle` as:

```
ax5SS: "[| [ $H, T:A0, F:A1, $E ] ;
           [ $H, F:~(A0), T:~(A1), $E ] |]
           ==> [$H, F:A0 --> A1, $E]"
```

The proof that `ax5` and `ax5SS` are indeed equivalent tableau rules, in the sense that one can be derived from the other in the presence of the remaining rules of our theory, can now be done directly with the help of `Isabelle`'s meta-logic. The rules mentioned below are the ones listed in the above theory. In what follows, verbatim text prefixed by `>` indicates user input entered at `Isabelle`'s command line environment.

First we prove that `ax5SS` can be derived from the axioms of our theory.

```
> Goal " [| [ $H, T:A0, F:A1, $G ] ;
            [ $H, F:~(A0), T:~(A1), $G ] |]
            ==> [$H, F:A0 --> A1, $G]";
```

Let `A0` and `A1` be represented by the schemas α and β and the contexts `$H` and `$G` be represented by Γ and Δ. We start thus to construct our tableau from the sequence $\Gamma, F{:}\alpha \rightarrow \beta, \Delta$ and intend to extend it into the two branches $\Gamma, T{:}\alpha, F{:}\beta, \Delta$ and $\Gamma, F{:}\neg\alpha, T{:}\neg\beta, \Delta$.

Notice that we can apply rule `ax5` over the initial sequence:

```
> by (resolve_tac [ax5] 1);
```

The following three branches originate then from $\Gamma, F{:}\alpha \rightarrow \beta, \Delta$, as new sub-goals:

[1] $\Gamma, F{:}\alpha, F{:}\neg\alpha, F{:}\beta, T{:}\neg\beta, \Delta$

[2] $\Gamma, T{:}\alpha, F{:}\beta, T{:}\neg\beta, \Delta$

[3] $\Gamma, T{:}\alpha, F{:}\beta, F{:}\neg\beta, \Delta$

To obtain the two initially intended branches from those, we may now just apply the *thinning* structural rule:

```
> by (res_inst_tac [("A","<<F:A0>>"),("B","<<F:A1>>")] thin 1);
```

Notice how that forces the instantiation of the sequences `$A` and `$B`, respectively, with the singleton sequences constituted of the signed formulas `F:A0` and `F:A1`. The new version of subgoal [1.1] that results from that transformation is:

[1.1] $\Gamma, F{:}\neg\alpha, T{:}\neg\beta, \Delta$

Similar transformations can be applied to subgoals [2] and [3]:

```
> by (res_inst_tac [("A","<<T:~A1>>")] thin 2);
> by (res_inst_tac [("A","<<F:~A1>>")] thin 3);
```

This originates, of course:

[2.1] $\Gamma, T{:}\alpha, F{:}\beta, \Delta$

[3.1] $\Gamma, T{:}\alpha, F{:}\beta, \Delta$

Notice how [2.1] and [3.1] coincide with the second branch of `ax5SS` and [1.1] corresponds to its first branch. The proof will be finished thus if one identifies the latter subgoals 'by assumption' with the intended branches that were entered as premises of the initial `Goal` command:

```
> by (REPEAT (assume_tac 1));
```

The message `No subgoals!`, issued by `Isabelle`, indicates that the proof is done. We can now record the result under the name `ax5SS`:

```
> qed "ax5SS";
```

Conversely, in the next step, we will prove `ax5` as a derived rule of our tableau system from the remaining axioms, together with `ax5SS`.

```
> Goal "[| [ $H, F:A0, F:~(A0), F:A1, T:~(A1), $G ] ;
           [ $H, T:A0, F:A1, T:~(A1), $G ] ;
           [ $H, T:A0, F:A1, F:~(A1), $G ] |]
           ==> [ $H, F:A0 --> A1, $G ]";
```

This time one can apply rule `ax5SS` over the initial sequence, $\Gamma, F{:}\alpha \to \beta, \Delta$:

```
> by (resolve_tac [ax5SS] 1);
```

The following two branches originate then as new subgoals:

[1] $\Gamma, T{:}\alpha, F{:}\beta, \Delta$
[2] $\Gamma, F{:}\neg\alpha, T{:}\neg\beta, \Delta$

We may apply `ax0` to subgoal [2]:

```
> by (resolve_tac [ax0] 2);
```

That will transform subgoal [2] into the new branch:

[2.1] $\Gamma, F{:}\neg\alpha, F{:}\beta, T{:}\neg\beta, \Delta$

Next, we may apply `ax4` to [2.1]:

```
> by (resolve_tac [ax4] 2);
```

This will transform [2.1] into the two new branches:

[2.1.1] $\Gamma, F{:}\alpha, F{:}\neg\alpha, F{:}\beta, T{:}\neg\beta, \Delta$
[2.1.2] $\Gamma, T{:}\alpha, F{:}\beta, T{:}\neg\beta, \Delta$

This time thinning alone won't do to finish the proof, though, as more branches and formulas will be needed to emulate `ax5`. It will be helpful thus to apply `axC4` to subgoal [1], or else `axC1` with instantiation over the cut-formula `A`.

```
> by (resolve_tac [axC4] 1);
```

From that we know that subgoal [1] will be transformed into the two new branches:

[1.1] $\Gamma, T{:}\alpha, F{:}\beta, T{:}\neg\beta, \Delta$
[1.2] $\Gamma, T{:}\alpha, F{:}\beta, F{:}\neg\beta, \Delta$

Observe that [1.1] coincides with [2.1.2], an intended branch of `ax5`, and the two other branches are provided by [1.2] and [2.1.1]. Thus, we can finish the proof using the premises:

```
> by (REPEAT (assume_tac 1));
```

With that we have proven that the two tableau systems for $Ł_3$, one with `ax5` and the other one with `ax5SS` in its place, are *equivalent*. The above proofs could of course have been performed, more appropriately, directly inside a theory obtained by erasing axiom `ax5` from the above theory `TL3`.

One last observation. For each symbol that does not appear in `Connectives` and `Primitives` a definition by abbreviation is expected in `CDef`. In that case one could use a strategy similar to the one above to propose and prove in `Isabelle` a list of derived rules involving that connective symbol.

5   Epilogue

The present paper has reported on the first concrete implementation of a certain constructive procedure for obtaining adequate two-signed tableau systems for a large number of finite-valued logics. Expressing a variety of logics in the *same* framework is quite useful for the development of comparisons between such logics, including their expressive and deductive powers. The general algorithm for tableaux for finite-valued logics proposed in [5], despite being more generally applicable in that it does not require the input logics to be 'sufficiently expressive', produces tableau rules having as many signs as the number of truth-values in the original semantical presentation of the given logic. Thus, for instance, even though a logic such as Łukasiewicz's $Ł_5$ is a deductive fragment of $Ł_3$ (in general, $Ł_m$ is a deductive fragment of $Ł_n$ iff $n - 1$ is a divisor or $m - 1$; check ch.8.5 of [6]), they will be hardly comparable inside such a multi-signed framework. In contrast, in our present framework, all two-signed rules of $Ł_5$ will be easily provable inside of $Ł_3$. We have indeed shown in the last section how it is already possible, with the help of the theories presently produced by our program, to use `Isabelle`'s meta-logic to show that two different axiomatizations for the same logic are equivalent. To show equivalence between axiomatizations from without the local perspective of a given theory, `Isabelle`'s *locales* (cf. [2]) will probably provide a better framework, and we hope to deal with that in the next version of our program.

There still remains some room for improvement and extension of both our algorithm (which should still, among other things, be upgraded in order to deal in general with first-order truth-functional logics) and its implementation. By way of an example, we have assumed from the start that the logics received as inputs to our program came together with a suitable collection of separating connectives. This second input, however, could be dispensed with, as the set of all definable unary connectives can in fact be automatically generated in finite time from any given initial set of operators of the input logic. That generation, however, may be costly for logics with a large number of truth-values and is not as yet performed by our program. Another direction that must be better explored, from the theoretical perspective, concerns the conditions for the admissibility or at least for the explicit control of the application of the dual-cut

14

rule. On the one hand, the elimination of dual-cut has an obvious favorable effect on the definition of completely automated theorem-proving tacticals for our logics. If that result cannot be obtained in general but if we can at least guarantee, on the other hand, that this dual-cut rule will never be needed, in each case, for more than a finite number of known formulas —say, the ones related to the original goal as constituting its subformulas or being the result of applying the separating connectives to its subformulas— then again this will make it possible to devise tacticals for obtaining fully automated derivations using the above described tableaux for our finite-valued logics. An important recent advance in that direction has in fact been done in [3], where the axiom extraction algorithm has already been upgraded in order to originate entirely analytic tableau systems, without the dual-cut rule or any potentially 'complexifying' rules such as `axC3` and `axC4`. A future version of our program had better be adapted to this new procedure, which will allow, moreover, for the algorithmic generation of fully automated proof tacticals, easily expressible in the framework of `Isabelle`.

Acknowledgments

References

[1] Matthias Baaz, Christian G. Fermüller, and Gernot Salzer. Automated deduction for many-valued logics. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1355–1402. Elsevier and MIT Press, 2001.

[2] Clemens Ballarin. Interpretation of locales in `Isabelle`: Theories and proof contexts. In J. M. Borwein and W. M. Farmer, editors, *Mathematical Knowledge Management*, (MKM 2006), LNAI 4108, pages 31–43. Springer, 2006.

[3] Carlos Caleiro and João Marcos. Classic-like analytic tableaux for finite-valued logics. 2009. Preprint available at:
`http://wslc.math.ist.utl.pt/ftp/pub/CaleiroC/09-CM-ClATab4FVL.pdf`.

[4] Carlos Caleiro, Walter Carnielli, Marcelo E. Coniglio, and João Marcos. Two's company: "The humbug of many logical values". In J.-Y. Béziau, editor, *Logica Universalis*, pages 169–189. Birkhäuser Verlag, Basel, Switzerland, 2005.
`http://wslc.math.ist.utl.pt/ftp/pub/CaleiroC/05-CCCM-dyadic.pdf`.

[5] Walter A. Carnielli. Systematization of the finite many-valued logics through the method of tableaux. *The Journal of Symbolic Logic*, 52(2):473–493, 1987.

[6] Roberto L. Cignoli, Itala M. L. D'Ottaviano and Daniele Mundici. *Algebraic Foundations of Many-Valued Reasoning*, volume 7 of *Trends in Logic*. Dordrecht: Kluwer, 1999.

[7] Reiner Hähnle. Tableaux for many-valued logics. In M. D'Agostino et al., editors, *Handbook of Tableau Methods*, pages 529–580. Springer, 1999.

[8] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. `Isabelle/HOL` — *A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[9] Raymond M. Smullyan. *First-Order Logic*. Dover, 1995.

[10] Wojciech Suchoń. La méthode de Smullyan de construire le calcul n-valent de Łukasiewicz avec implication et négation. *Reports on Mathematical Logic*, 2:37–42, 1974.

[11] Heinrich Wansing and Yaroslav Shramko. Suszko's Thesis, inferential many-valuedness, and the notion of a logical system. *Studia Logica*, 88(3):405–429, 2008.