

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 tableaux rules for a large class of finite-valued logics. Its output consists of `Isabelle` theories.¹

Key words: Many-valued logics, tableaux, automated theorem proving.

1 Introduction

This note will report on the first developments towards the implementation of a fully automated system 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 [2]. Surveys on tableaux for many-valued logics can be found in [4, 1]. The implementation has been performed in ML, and its application gives rise to an `Isabelle` theory (check [5]) formalizing a given finite-valued logic in terms of two-signed tableau rules.

The survey paper [4] 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 results concerning many-valued logics as well as their homologous presentation in terms of bivalent semantics defined 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 \mathbb{L}_3 , the well-known 3-valued logic of Łukasiewicz.

The 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, for the case of \mathbb{L}_3 .

¹ A snapshot of the corresponding code can be checked in <http://tinyurl.com/5cakro>.

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

2 Many-valued logics

Given a denumerable set At of *atoms* and a finite family $\text{Cct} = \{\odot_j^i\}_{j \in J}$ of *connectives*, where $\text{arity}(\odot_j^i) = i$, 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 collection of mappings $\{\xi_k^\nu\}_{k \in K}$ where $\text{dom}(\xi_k^\nu) = \mathcal{S}$ and $\text{codom}(\xi_k^\nu) = \mathcal{V}_k$, 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 ξ_k^ν themselves may be called (ν -valued) *valuations*, where $\nu = \text{Card}(\mathcal{V}_k)$. A *bivalent semantics* is any semantics where \mathcal{D}_k and \mathcal{U}_k are singleton sets, for any $k \in K$. For bivalent semantics, valuations are often called *bivaluations*.

The canonical notion of (single-conclusion) *entailment* $\models_{\text{Sem}} \subseteq \text{Pow}(\mathcal{S}) \times \mathcal{S}$ induced by a semantics Sem is defined by setting $\Gamma \models_{\text{Sem}} \varphi$ iff $\xi_k^\nu(\varphi) \in \mathcal{D}_k$ whenever $\xi_k^\nu(\Gamma) \subseteq \mathcal{D}_k$, for every $\xi_k^\nu \in \text{Sem}$. The pair $\langle \mathcal{S}, \models_{\text{Sem}} \rangle$ may be called a *generic ν -valued logic*, where $\nu = \text{Max}_{k \in K}(\text{Card}(\mathcal{V}_k))$.

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

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, consider $\mathcal{V}_2 = \{T, F\}$ and $\mathcal{D}_2 = \{T\}$, and consider the ‘binary print’ of the algebraic truth-values produced by the total mapping $t : \mathcal{V} \rightarrow \mathcal{V}_2$, defined by $t(v) = T$ iff $v \in \mathcal{D}$. For any ν -valuation ξ^ν of a given semantics Sem , consider now the characteristic total function $b_\xi = t \circ \xi^\nu$. Now, collect all such bivaluations b_ξ ’s into a new semantics $\text{Sem}(2)$, and note that $\Gamma \models_{\text{Sem}(2)} \varphi$ iff $\Gamma \models_{\text{Sem}} \varphi$. The standard 2-valued notion of inference of Classical Logic is characterized indeed by a bivalent truth-functional semantics. In general, though, a bivalent characterization of a logic with a ν -valued truth-functional semantics explores the trade-off between the ‘algebraic perspective’ of many-valuedness, with its many ‘algebraic truth-values’ and its semantic characterization in terms of a set of homomorphisms, on the one hand, and the classic-inclined ‘logical perspective’, with its emphasis on characterizations based on 2 ‘logical values’,

on the other hand (for more detailed discussions of this issue, check [2, 7]). 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 this paper will involve Łukasiewicz's well-known 3-valued logic L_3 , characterized by the algebra of truth-values $\{1, \frac{1}{2}, 0\}$, $\{1\}$, $\{\neg, \rightarrow, \vee, \wedge\}$, 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 $(v_1 \rightarrow v_2) = \text{Min}(1, 1 - v_1 + v_2)$. The binary symbols \vee and \wedge can be introduced as primitive interpreting them through $(v_1 \vee v_2) = \text{Max}(v_1, v_2)$ and $(v_1 \wedge v_2) = \text{Min}(v_1, v_2)$, but they can also more simply be introduced by definition just like in Classical Logic, 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 print of an arbitrary atom of L_3 and of its negation is illustrated in the table below.

v	$t(v)$	$\neg v$	$t(\neg v)$
1	T	0	F
$\frac{1}{2}$	F	$\frac{1}{2}$	F
0	F	1	T

(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 n -valued operation, for $n = \text{Card}(\mathcal{V})$, may be defined with the help of a suitable combination of its primitive operators $\{\hat{\odot}_j^i\}_{j \in J}$. When \mathcal{L} is not functionally complete from the start, we may consider \mathcal{L}^{fc} as any functionally complete n -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 $v_1 \# v_2$, 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 $\hat{\odot}$ of a given truth-functional logic, with interpretation $\hat{\odot}$, we say that $\hat{\odot}$ *separates* v_1 and v_2 in case $\hat{\odot}(v_1) \# \hat{\odot}(v_2)$. Obviously, for any pair of truth-values of \mathcal{L}^{fc} it is possible to find or to define in the corresponding term algebra an appropriate *separating connective* $\hat{\odot}$. 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, how the negation connective of L_3 separates the two undesigned truth-values. Based on Table 1, one may in fact easily provide a unique identification to each of the 3 initial algebraic truth-values, by way of the following statements:

$$\begin{aligned}
 v = 1 & \text{ iff } t(v) = T & (I) \\
 v = \frac{1}{2} & \text{ iff } t(v) = F \text{ and } t(\neg v) = F \\
 v = 0 & \text{ iff } t(v) = F \text{ and } t(\neg v) = T
 \end{aligned}$$

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

$$\begin{array}{c}
\begin{array}{|c|c|c|c|} \hline A & 1 & \frac{1}{2} & 0 \\ \hline 1 & 1 & \frac{1}{2} & 0 \\ \hline \frac{1}{2} & 1 & 1 & \frac{1}{2} \\ \hline 0 & 1 & 1 & 1 \\ \hline \end{array}
\quad
\begin{array}{|c|c|c|c|} \hline B & 1 & \frac{1}{2} & 0 \\ \hline 1 & 0 & \frac{1}{2} & 1 \\ \hline \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ \hline 0 & 0 & 0 & 0 \\ \hline \end{array}
\end{array} \tag{2}$$

From those tables it is clear for instance that:

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

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

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

In [2] 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 define a notion of entailment that coincides with the original entailment defined with the help of the algebra of truth-values \mathcal{TV} . Moreover, all those clauses are in a specific format we call *gentzenian*, namely, they are conditional expressions of the form $(\Phi \Rightarrow \Psi)$ where both Φ and Ψ are (meta)formulas of the form \top (top), \perp (bottom) or a clause of the form

$b(\varphi_1^1) = w_1^1 \& \dots \& b(\varphi_1^{n_1}) = w_1^{n_1} \mid \dots \mid b(\varphi_m^1) = w_m^1 \& \dots \& b(\varphi_m^{n_m}) = w_m^{n_m}.(G)$
Here, $w_i^j \in \{T, F\}$, each φ_i^j is a formula of \mathcal{L} , the symbol \Rightarrow represents implication (and \Leftrightarrow shall represent bi-implication, abbreviating the conjunction of two clauses of the form (G)), the symbol $\&$ represents conjunction, and \mid represents disjunction. The (meta)logic governing these clauses is FOL, First-Order Classical Logic. One may alternatively represent a clause of the form (G) as $\bigvee_{1 \leq k \leq m} \bigwedge_{1 \leq s \leq n_m} (b(\varphi_k^s) = w_k^s)$.

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

$$T : \neg(\alpha \rightarrow \beta) \Leftrightarrow T : \alpha \& F : \beta \& T : \neg\beta \tag{IV}$$

Following the above line of reasoning, it is also correct to write, for instance, the clause:

$$\begin{array}{l}
F : (\alpha \rightarrow \beta) \Leftrightarrow T : \alpha \& F : \beta \& F : \neg\beta \mid \\
T : \alpha \& F : \beta \& T : \neg\beta \mid \\
F : \alpha \& F : \neg\alpha \& F : \beta \& T : \neg\beta
\end{array} \tag{V}$$

According to the reductive algorithm described in [2], a sound and complete bivalent version of any sufficiently expressive finite-valued logic \mathcal{L} is obtained if:

(i) the above illustrated procedure is iterated in order to obtain clauses describing exactly in which situation one may assert $T : \mathbb{C}_j^i(\alpha_1, \dots, \alpha_i)$ and $F : \mathbb{C}_j^i(\alpha_1, \dots, \alpha_i)$, as well as $T : \mathbb{S}\mathbb{C}_j^i(\alpha_1, \dots, \alpha_i)$ and $F : \mathbb{S}\mathbb{C}_j^i(\alpha_1, \dots, \alpha_i)$, for each $\mathbb{C}_k^i \in \text{Cct}$ and each one of the separating connectives \mathbb{S} of \mathcal{L} ;

(ii) to all those clauses one adds the following extra axioms governing the behavior of the admissible collection of bivaluations:

$$\begin{aligned} \text{(C1)} \quad \top &\Rightarrow T : \alpha \mid F : \alpha & \text{(C2)} \quad T : \alpha \ \& \ F : \alpha &\Rightarrow \perp \\ \text{(C3)} \quad T : \alpha &\Rightarrow \bigvee_{d \in \mathcal{D}} \bigwedge_{1 \leq m < n \leq \text{Card}(\mathcal{D})} w_{mn}^d : \mathbb{S}_{mn}(\alpha) \\ \text{(C4)} \quad F : \alpha &\Rightarrow \bigvee_{u \in \mathcal{U}} \bigwedge_{1 \leq m < n \leq \text{Card}(\mathcal{U})} w_{mn}^u : \mathbb{S}_{mn}(\alpha) \end{aligned}$$

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

3 Tableaux

Generic tableau systems for finite-valued logics are known at least since [3]. 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. But 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 [2], 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, as the following tableau rules:

$$\begin{array}{ccc} \text{(IV)}^{tab} & T : \neg(\alpha \rightarrow \beta) & F : (\alpha \rightarrow \beta) & \text{(V)}^{tab} \\ & \mid & \diagup \quad \mid \quad \diagdown & \\ & T : \alpha & T : \alpha & T : \alpha & F : \alpha \\ & F : \beta & F : \beta & F : \beta & F : \neg\alpha \\ & T : \neg\beta & F : \neg\beta & T : \neg\beta & F : \beta \\ & & & & T : \neg\beta \end{array}$$

To those rules corresponding to the truth-tables of the operators and the separating connectives, one should also add rules corresponding to the extra axioms (C1)–(C4). In practical cases, however, axioms (C3) and (C4) can often be 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’ *branching rule*, for arbitrary α :

$$\begin{array}{c} \diagup \quad \diagdown \\ T : \alpha \quad F : \alpha \end{array}$$

All other definitions and concepts concerning the construction of tableaux are standard (check [6]).

One might be worried, with good reason, that the unrestrained use of the branching rule may potentially make the corresponding tableaux non-analytic. We will discuss that in the conclusion. The tableau rules originated from 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 $F : (\alpha \rightarrow \beta)$ can eventually be simplified into one of the following equivalent two-branching rules:

$$\begin{array}{ccc}
 (V^*)^{tab} & F : (\alpha \rightarrow \beta) & (V^{**})^{tab} \\
 & \swarrow \quad \searrow & \swarrow \quad \searrow \\
 T : \alpha & F : \alpha & T : \alpha & F : \neg\alpha \\
 F : \beta & F : \neg\alpha & F : \beta & T : \neg\beta \\
 & F : \beta & & \\
 & T : \neg\beta & &
 \end{array}$$

4 Implementation

We used the functional programming language ML to automate the axiom extraction process. ML provides us, among other advantages, with an elegant and suggestive syntax, and a very handy compile-time type checking and type inference that guarantees that we never run into unexpected run-time problems with our program, once it is proved correct with respect to the specification.

The relevant inputs of our program include the detailed definition of a finite-valued logic, such as the logic L_3 presented in the previous sections, together with an appropriate set of separating connectives for that logic.

Here's an example of a input for the logic L_3 , presented above, where the functions `CSym`, `CAri` and `CTab` take a connective and return its symbol (for printing), arity and truth-table, respectively. A truth-table of a given connective \odot is represented as the list of all pairs $([x_1, \dots, x_n], y)$ such that $\odot(x_1, \dots, x_n) = y$.

```

(* PROGRAM SIGNATURE *)          (* EXAMPLE OF L3 *)
signature LOGIC =
sig
  type cct
  val Values
  val Designated
  val Connectives
  val SeparatingD
  val SeparatingND
  val CSym
  val CAri

  datatype cct = Neg | Imp
  ["0", "1/2", "1"];
  ["1"];
  [Neg, Imp];
  [];
  [Neg];
  fun CSym Neg = "~"
    | CSym Imp = "-->";
  fun CAri Neg = 1
    | CAri Imp = 2;

```

```

val CTab
    fun CTab Neg = [ ([ "0", "1"],
                      ([ "1/2", "1/2"],
                       ([ "1", "0" ]
                        | CTab Imp = [ ([ "0", "0"], "1"),
                                       ([ "0", "1/2"], "1"),
                                       ([ "0", "1"], "1"),
                                       ([ "1/2", "0"], "1/2"),
                                       (...),
                                       ([ "1", "1"], "1" ) ]
end;

```

To perform the extraction, our program first generates a list of all necessary rules, as explained in section 2.

```

val rulesList = [T "~(AO)",      T "AO --> A1",
                 T "~(~(AO))",  T "~(AO --> A1)",
                 F "~(AO)",      F "AO --> A1",
                 F "~(~(AO))",  F "~(AO --> A1)" ];

```

Next, the program converts each connective's truth-table, here given by the function `CTab`, 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 `SeparatingND` for undesignated values). For instance, for the case of the connective `Neg`, the clauses are the following:

```

(* AO *)          (* ~(AO) *)
([F "AO",T "~(AO)"], [T "~(AO)"])
([F "AO",F "~(AO)"], [F "~(AO)",F "~(~(AO))"])
([T "AO"],          [F "~(AO)",T "~(~(AO))"])

```

Now, for each formula in the list of rules, a search is performed through all tables generated in the latter step, and all clauses in which the given formula appears on the right hand side are returned. The left hand side of these clauses represents the branches of the desired tableau rule. For instance, the rules for $T:\sim A$ and $F:\sim A$ are:

```

(* T:~A *)      ([ [F "AO",T "~(AO)"] ],          T "~(AO)")
(* F:~A *)      ([ [F "AO",F "~(AO)"], [T "AO"] ],  F "~(AO)")

```

The next steps include the calculus of axioms (C3) and (C4), and the printing of all definitions, concrete syntax and rules into a text file containing the full theory ready to use in `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 theories with rules and axioms for various kinds of deductive formalisms, and equally straightforward to define tacticals for the automation of routine tasks and to prove theorems about these systems.

For the case of L_3 , here is the corresponding theory produced as output by our program:

```

theory TL3

imports Sequents
begin
typedecl a
consts
  Trueprop :: "(seq'=>seq') => prop"
  True      :: o
  False     :: o
  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)

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

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

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

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

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

ax4: "[| [ $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 ]"

```

```

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

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

ML {* use_legacy_bindings (the_context ()) *}
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 signed formula (typed `o`). We also extract from the logic received as input each constructor's name to use as syntactic sugar, as well as its associativity rules and priority order.

In the generated axioms corresponding to the tableau rules, `T:X` and `F:X` are (signed) formulas, `$H` and `$E` are sequences of such formulas (contexts) that are not directly involved in the rule, 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. In Isabelle, the application of a rule means that it is 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 at each step. The branching rule corresponds to axiom `axC1`, and the closure rule for a branch of the tableau corresponds to the axioms `axC21` and `axC22`.

Note for instance that `ax5` corresponds to rule $(V)^{tab}$ from the last section. The simpler rule $(V^{**})^{tab}$, mentioned in the same section, can of course 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 `ax5S` are indeed equivalent tableau rules, i.e., that one can derive one 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. One might notice that, while in [3] the number of rules for each given primitive connective is exponential in its arity, here the number of rules for each such connective is always polynomial both in the arity and in the number of separating connectives of the input logic. The worst-case number of nodes involved in each tableau rule, however, using our algorithm, is exponential in the arity of the corresponding connective. Using Isabelle's meta-logic to prove simplifications such as the one illustrated above, for the case of rule (V) , the number of nodes involved in each tableau rule may be substantially reduced.

5 Epilogue

The present note 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.

There still remains some room for improvement and extension of both our algorithm (which should still, for instance, 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 system. 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 branching 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 branching 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.

References

1. Matthias Baaz, Christian G. Fermüller, and Gernot Salzer. Automated deduction for many-valued logics. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1355–1402. Elsevier and MIT Press, 2001.
2. 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.
URL = <http://wslc.math.ist.utl.pt/ftp/pub/CaleiroC/05-CCCM-dyadic.pdf>.
3. 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.
4. Reiner Hähnle. Tableaux for many-valued logics. In M. D’Agostino et al., editors, *Handbook of Tableau Methods*, pages 529–580. Springer, 1999.
5. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
6. Raymond M. Smullyan. *First-Order Logic*. Dover, 1995.
7. 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.