# Modifications on the implementation of a framework for tableaux methods

Adolfo Gustavo Serra Seca Neto

July 1, 2003

**Abstract**

The method of tableaux is a formal proof procedure with many varieties and for several logics. Among other important applications, it is used for the development of theorem-provers. An object-oriented framework for the implementation of tableaux methods was presented in Wagner Dias's MSc. Dissertation [Dia02]. This framework was implemented in the C++ programming language. In this paper, we describe some modifications we have made in that implementation in order to obtain shorter KE Tableau proofs. We also present the result of tests we have run with both implementations comparing: *(i)* the number of formulas, *(ii)* the number of nodes in the closed tableaus and *(iii)* the time spent for the closing of each tableau.

# 1 Introduction

The method of tableaux is a formal proof procedure with many varieties and for several logics. Among other important applications, it is used for the development of theorem-provers. An object-oriented framework for the implementation of tableaux methods was presented in Wagner Dias's MSc. Dissertation [Dia02]. This framework was implemented in the C++ programming language. Three tableaux methods were covered by his implementation: Smullyan's Analytic Tableaux, D'Agostino's KE Tableaux and Finger and Wasserman's KE-$S_3$ Tableaux. In this paper, our goal is describing some modifications we have made in that implementation in order to obtain shorter KE Tableau proofs, as well as presenting the result of tests we have run with both implementations comparing: *(i)* the number of formulas, *(ii)* the number of nodes in the closed tableaus and *(iii)* the time spent for the closing of each tableau.

In Section 2, we shortly present some of the history of tableaux, give a precise definition of Analytic Tableaux, and recall some of the considerations about tableaux made by Marcello D'Agostino in [DAg99]. After that, we present his KE System in subsection 3.1. Then in Section 4, we show the modifications we have made in Wagner Dias's system. After that, in Section 5, we introduce the families of formulas that were used for the comparative tests whose results are in Section 6. We present in Section 7 our conclusions and possible future works. And in Appendix A, we give a personal evaluation on the experience of rewriting code.

# 2 What are Tableaux?

According to Smullyan [Smu68], Analytic Tableaux are "an extremely elegant and efficient proof procedure for propositional logic". It is a variant of the *"semantic tableaux"* of Beth[Bet59], or of methods of Hintikka[Hin55]. Ultimately, the whole idea derives from Gentzen [Gen35].

The method of tableaux is a formal proof procedure with many varieties and for several logics. It is a refutation procedure — that is, in order to prove that a formula $X$ is valid we try to show that it is not valid. With this in mind, we apply a procedure to infer the logical consequences of the formulas present in the tableau. If some branch remains open, the we have a refutation of $X$.

## 2.1 The history of tableaux

In [Fit99], tableau history is analysed. It began with Gentzen, who introduced two formal proof procedures, sequent calculus and natural deduction, in 1935. Gentzen showed that his sequent calculus had a rule, called Cut, which could be eliminated. He then proved the famous Cut elimination theorem.

In 1955, Beth, motivated by semantic concerns, introduced the 'semantic tableaus'. Hintikka, simultaneously, also motivated by semantic concerns, that is, "the idea behind a proof of $X$ is that it is a systematic attempt to construct a model in which $\neg X$ is true, if the attempt fails, $X$ has been established as valid", introduced *model sets*.

But it was with Smullyan [Smu68] that tableaus became widely known. In his book First-Order Logic he presents a system called 'Analytic[1] Tableaux'.

## 2.2 Precise Definition of Tableaux

In order to give a precise definition of Tableaux, Smullyan first define trees, formulas and some other associated concepts. Below we present some of these definitions taken from [Smu68].

### 2.2.1 Trees

An *unordered tree* consists of three mathematical objects:

1. A set $S$ of elements called *points*
2. A function $l$, which assigns to each point $x$ a positive integer $l(x)$ called the *level* of $x$
3. A relation $xRy$ defined in $S$, which we read as "$x$ is a *predecessor* of $y$" or "$y$ is a *successor* of $x$". This relation must obey the following conditions:

   $C_1$: There is a unique point $a_1$ of level 1. This point is called the *origin* of the tree.
   $C_2$: Every other point besides the origin has exactly one predecessor.
   $C_3$: For any points $x$, $y$, if $y$ is a successor pof $x$, then $l(y) = l(x) + 1$.

An *end point* is a point which has no successors. A *simple point* has exactly one successor and a *junction point* has more than one successor. A *path* is any finite or denumerable sequence of points, beginning with the origin, which is such that each term of the sequence (except the last, if there is one) is the predecessor of the next. And a *maximal path* or *branch* is either a path whose last term is an end point of the tree, or a path which is infinite.

An *ordered tree* is an unordered tree along with a function $\theta$ which assigns to each junction point $z$ a sequence $\theta(z)$ with no repetitions, and whose set of terms consists of all the sucessors of $z$. Therefore, if $z$ is a junction point of an ordered tree, we can speak of the $1^{st}$, $2^{nd}$, ... , $n^{th}$, ... successors of $z$ (for any $n$ up to the number of successors of $z$) meaning, of course, the $1^{st}$, $2^{nd}$, ... , $n^{th}$, ... terms of $\theta(z)$. For a simple point $x$, we shall also speak of the successor of $x$ as the *sole* successor of $x$.

Given a tree $T$, we can add "new" points as successors of an end point $x$ of this tree. That is, for any element $y$ not belonging to $T$, the adjunction of $y$ as the sole successor of $x$ results in the tree obtained by:

1. adding $y$ to the set $S$,
2. adding the ordered pair $< x, y >$ to the relation $R$ (looked as a set of ordered pairs), and
3. extending the function $l$ by defining $l(y) = l(x) + 1$.

The above description serves for adding one point. If we want to add more than one point, that is, if we want to add the distinct elements $y_1, \ldots, y_n$, each outside $S$, as respective $1^{st}$, $2^{nd}$, ... , $n^{th}$ successors of $x$, we will have to:

1. add the $y_i$ to $S$,
2. add the pairs $< x, y_i >$ to $R$,
3. extend $l$ by setting $l(y_1) = \ldots = l(y_n) = l(x) + 1$, and
4. extend the function $\theta$ by defining $\theta(x)$ to be the sequence $(y_1, \ldots, y_n)$.

A tree is called *finitely generated* if each point has only finitely many successors. A tree, $T$, is called *finite* if $T$ has only finitely many points, otherwise the tree is called *infinite*. Obviously, a finitely generated tree may be infinite.

A *dyadic tree* is an ordered tree in which each junction point has exactly 2 successors. For such trees, the first successor of a junction point is called the *left successor*, and the second successor is the *right successor*.

[1]Analytic because the subformula property is a central feature.

### 2.2.2 Formulas of Propositional Logic

The *logical connectives* are represented by the following four symbols:

1. $\neg$: the negation (read "not"),
2. $\wedge$: the conjunction (read "and"),
3. $\vee$: the disjunction (read "or"),
4. $\rightarrow$: the implication (read "implies").

The first sysmbol is the *unary* connective whilst the last three are collectively called *binary* connectives. Other symbols are:

- A denumerable set $p_1, p_2, \ldots, p_n, \ldots$ of symbols called *propositional variables.* Usually the letters "$p$", "$q$", "$r$", "$s$" stand for any of the variables $p_1, p_2, \ldots, p_n, \ldots$
- The two symbols (, ), respectively called the *left parenthesis* and the *right parenthesis*, which are used for punctuation.

Then, a *formula* can be recursively defined from the following set of rules:

$F_0$ Every propositional variable is a formula.

$F_1$ If $A$ is a formula, $\neg A$ is also a formula.

$F_2$ If $A, B$ are formulas, $(A \wedge B)$ is also a formula.

$F_3$ If $A, B$ are formulas, $(A \vee B)$ is also a formula.

$F_4$ If $A, B$ are formulas, $(A \rightarrow B)$ is also a formula.

A *formation sequence* is any finite sequence such that each term of the sequence is either a propositional variable or is of the form $\neg A$, where $A$ is an earlier term of the sequence, or is of one of the forms $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, where $A, B$ appeared before in the sequence. Then we define $A$ to be a *formula* if there exists a formation sequence whose last term is $A$. And such a sequence is called a *formation sequence* for $A$.

For any formula $A$, $\neg A$ is the *negation* of $A$. And for any two formulas $A, B$, we say that $(A \wedge B)$, $(A \vee B)$ and $(A \rightarrow B)$ are the *conjunction*, *disjunction* and *conditional* of $A, B$ respectively. In a conditional formula, $(A \rightarrow B)$, $A$ is called the *antecedent* and $B$ the *consequent.*

Particularly for tableaux, Smullyan presents signed formulas. A *signed formula* is an expression $TX$ or $FX$ where $X$ is a (unsigned) formula and the symbols $T$ and $F$ are the truth-values true and false.

The *conjugate* of a signed formula is the result of changing "$T$" to "$F$" or "$F$" to "$T$" (thus teh conjugate of $T\,X$ if $F\,X$; the conjgate of $F\,X$ is $T\,X$).

The method of tableaux is based on the fact that [Smu68] "under any interpretation the following eight facts hold (for any formulas $X, Y$):

1. (a) If $\neg X$ is true, then $X$ is false.
   (b) If $\neg X$ is false, then $X$ is true.
2. (a) If a conjunction $X \wedge Y$ is true, then $X, Y$ are both true.
   (b) If a conjunction $X \wedge Y$ is false, then either $X$ is false or $Y$ is false.
3. (a) If a disjunction $X \vee Y$ is true, then either $X$ is true or $Y$ is true.
   (b) If a disjunction $X \vee Y$ is false, then both $X, Y$ are false.
4. (a) If $X \rightarrow Y$ is true, then either $X$ is false or $Y$ is true.
   (b) If $X \rightarrow Y$ is false, then $X$ is true and $Y$ is false."

After that the rules for each logical connective are presented according to the eight facts shown above (see Figure 1). These rules allow the expansion of tableaus, as will be seen in subsection 2.2.4.

### 2.2.3 $\alpha$ and $\beta$ Rules

Smullyan noticed similarities between some rules and classified them in two groups: formulas of conjunctive type ($\alpha$ rules, also known as *linear type* rules) and formulas of disjunctive type ($\beta$ rules, also known as *branching type* rules).

The formulas of *conjuntive* type are represented by the greek letter $\alpha$. He uses "$\alpha$" to stand for any signed formula of type $A$—i.e. of one of the five forms $T\,(A \wedge B), F\,(A \vee B), F\,(A \rightarrow B), T\,\neg A, F \neg A$. For every such formula, he *defines* the two formulas $\alpha_1$ and $\alpha_2$ as follows:

$$\begin{gathered}
\text{If } T\,(A \wedge B)\text{, then } \alpha_1 = T\,A \text{ and } \alpha_2 = T\,B \\
\text{If } F\,(A \vee B)\text{, then } \alpha_1 = F\,A \text{ and } \alpha_2 = F\,B \\
\text{If } F\,(A \rightarrow B)\text{, then } \alpha_1 = T\,A \text{ and } \alpha_2 = F\,B \\
\text{If } T\,\neg A\text{, then } \alpha_1 = F\,A \text{ and } \alpha_2 = F\,A \\
\text{If } F \neg A\text{, then } \alpha_1 = T\,A \text{ and } \alpha_2 = T\,A
\end{gathered}$$

This is summarized in the following table:

| $\alpha$ | $\alpha_1$ | $\alpha_2$ |
|---|---|---|
| $T\,(A \wedge B)$ | $T\,A$ | $T\,B$ |
| $F\,(A \vee B)$ | $F\,A$ | $F\,B$ |
| $T\,(A \rightarrow B)$ | $T\,A$ | $F\,B$ |
| $T\,\neg A$ | $F\,A$ | $F\,A$ |
| $F\,\neg A$ | $T\,A$ | $T\,A$ |

The letter $\beta$ is used to stand for formulas of *disjunctive* type. These are the formulas of type $B$—i.e. one of the three forms $F\,(A \wedge B), T\,(A \vee B), T\,(A \rightarrow B)$. For every such formula $\beta$, the two formulas $\beta_1, \beta_2$ are as in the following table:

| $\beta$ | $\beta_1$ | $\beta_2$ |
|---|---|---|
| $F\,(A \wedge B)$ | $F\,A$ | $F\,B$ |
| $T\,(A \vee B)$ | $T\,A$ | $T\,B$ |
| $T\,(A \rightarrow B)$ | $F\,A$ | $T\,B$ |

Therefore, all tableau rules can be resumed to the following two:

$$\frac{T\,\neg A}{F\,A}\;(T\neg) \qquad \frac{F\,\neg A}{T\,A}\;(F\neg)$$

$$\frac{T\,(A \wedge B)}{\begin{array}{c}T\,A\\ T\,B\end{array}}\;(T\wedge) \qquad \frac{F\,(A \wedge B)}{F\,A \mid F\,B}\;(F\wedge)$$

$$\frac{T\,(A \vee B)}{T\,A \mid T\,B}\;(T\vee) \qquad \frac{F\,(A \vee B)}{\begin{array}{c}F\,A\\ F\,B\end{array}}\;(F\wedge)$$

$$\frac{T\,(A \rightarrow B)}{F\,A \mid T\,B}\;(T \rightarrow) \qquad \frac{F\,(A \rightarrow B)}{\begin{array}{c}T\,A\\ F\,B\end{array}}\;(F \rightarrow)$$

Figure 1: Analytic tableau expansion rules

$$\text{A Rule:} \quad \frac{\alpha}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \qquad \text{B Rule:} \quad \frac{\beta}{\beta_1 \mid \beta_2}$$

As an example, let us see a tableau proof of $(A \wedge B) \rightarrow (A \vee B)$

$$\begin{array}{c} F\,(A \wedge B) \rightarrow (A \vee B) \\ T\,A \wedge B \\ F\,A \vee B \\ T\,A \\ T\,B \\ F\,A \\ X \end{array}$$

### 2.2.4 Precise Definition

Given the definitions above, Smullyan could present the following definition [Smu68]:

> An *analytic tableau* for $X$ is an ordered dyadic tree, whose points are (occurences of) formulas, which is constructed as follows. We start by placing $X$ at the origin. Now suppose $T$ is a tableau for $X$ which has already been constructed; let $Y$ be an end point. Then we may extend $T$ by either of the following two operations.
>
> (A) If some $\alpha$ occurs on the path $P_Y$, then we may adjoin either $\alpha_1$ or $\alpha_2$ as the sole successor of $Y$. (In practice, we usually successively adjoin $\alpha_1$ and then $\alpha_2$.)
>
> (B) If some $\beta$ occurs on the path $P_Y$, then we may simultaneously adjoin $\beta_1$ as the left successor of $Y$ and $\beta_2$ as the right successor of $Y$.
>
> The above inductive definition of tableau for $X$ can be made explicit as follows. Given two ordered dyadic trees $T_1$ and $T_2$, whose points are ocurrences of formulas, we call $T_2$ a *direct extension* of $T_1$ if $T_2$ can be obtained from $T_1$ by one application of the operation (A) or (B) above. Then $T$ is a tableau for $X$ if and only if there exists a finite sequence $(T_1, T_2, \ldots, T_n = T)$ such that $T_1$ is a 1-point tree whose origin is $X$ and such that for each $i < n$, $T_{i+1}$ is a direct extension of $T_i$.

A branch $\theta$ of a tableau for signed formulas is *closed* if it contains some signed formula and its conjugate. And $T$ is called closed if every branch of $T$ is closed. By a *proof* of an unsigned formula $X$ is meant a closed tableau for $F\,X$.

In [**?**], Smullyan proves the consistency and completenes of this system for propositional logic. Among other things, he uses Hintikka's downward saturated sets in his proof.

# 3 Considerations about Tableau Systems

In [Fit99] one can see that there are several kinds of tableau systems for different logics. He also points that tableau systems are a good basis for the development of theorem-provers, although sometimes they may not turn out to be the most efficient approach to automated theorem-proving. But since tableau systems are relatively easy to develop, they will always have an important role in automated theorem proving, even when theorem-provers are not based on, but only derived from them.

Marcello DAgostino in [DAg99] discusses some important considerations about tableau systems and presents a variant over Smullyan's Analytic Tableaux: the KE System.

He motivates his study by remembering that when Wittgenstein and Post independently devised (back in the 1920s) the well-known decision procedure based on the truth-tables, it was thought that the deducibility problem of classical propositional logic was solved. However, the truth-table method requires checking $2^n$ rows for a formula with $n$ different propositional variables. That is, it is a very inefficient algorithm, even for relatively small values of $n$.

Besides that, reinforcing the importance of the analysis of proof procedures associated with a logic, Gabbay stated in [Gab96] that:

"a logical system **L** is not just the traditional consequence relation $\vdash$, but a pair $(\vdash, \mathbf{S}_\vdash)$, where $\vdash$ is a mathematically defined consequence relation (i.e. a set of pairs $(\Delta, Q)$ such that $\Delta \vdash Q$) and $\mathbf{S}_\vdash$ is an algorithmic system for generating all those pairs. Thus, according to this definition, classical logic $\vdash$ perceived as a set of tautologies together with a Gentzen system $\mathbf{S}_\vdash$ is not the same as classical logic together with the two-valued truth-table decision procedure $\mathbf{T}_\vdash$ for it. In our conceptual framework, $(\vdash, \mathbf{S}_\vdash)$ is *not the same logic* as $(\vdash, \mathbf{T}_\vdash)$."

D'Agostino argues that Gentzen's natural deduction is not really 'natural', that is, it does not come 'as close to actual reasoning' as Gentzen wrote in 1935. According to D'Agostino, "natural deduction rules do not capture the classical meaning of the logical operators" but, instead, natural deduction rules are a reading of Heyting's explanations of the *constructive meaning* of the logical operators.

He shows that some proofs in the natural deduction system, such as that for one of the deMorgan's laws:

$$\neg(P \wedge Q) \rightarrow (\neg P \vee \neg Q)$$

are rather unnatural (See [DAg99], page 64).

D'Agostino also shows that some Cut free sequent proofs are very redundant as well as some tableau proofs. He proves that the tableau system and cut free sequent calculus are intrinsically redundant. Therefore, he concludes, "such systems are not natural for classical logic".

## 3.1 The KE System

D'Agostino states that tableau rules are not really classical since one of the two principles that form the basis of the classical notion of truth, the Principle of Bivalence, is not clearly present in the tableau system. The principle of bivalence (also known as the Principle of the Excluded Middle) states that every proposition is either true or false, and there are no other possibilities. On the other hand, the other principle, the principle of Non-contradiction, which asserts that no proposition can be true and false at the same time, is clearly embodied in the rule for closing a branch in a tableau system.

Later D'Agostino presents the KE System, a refutation system that, he argues, though "being close to the tableau method, is not affected by the anomalies of cut-free systems." KE-Tableaux are very similar to Analytic Tableaux because they also use signed formulas, have expansion rules with premises and conclusions and have the same rules for the construction of the initial tableau and the closing of the tableau.

D'Agostino proposed KE-Tableaux as an improvement, in the computaional sense, over Smullyan's Analytic Tableaux. He proved that a canonical restriction of the KE System can polynomially simulate the tableau method but the tableau method cannot polinomially simulate the canonical restriction of KE[2]. That is, the KE System is a 'natural' proof method and it is more efficient than Smullyan's tableaux.

The set of rules for the KE System (see Figure 2) has only one rule of the branching type, the PB rule, corresponding to the principle of bivalence. With this rule, in conjunction with the usual tableau rules, it is not difficult to construct short refutations of the 'hard examples' he showed for Smullyan's tableaux.

Besides that, the KE System introduces simple elimination rules of linear type which combined with PB yield a refutation system for classical logic. And he also requires that the system is restricted to analytic applications of PB, i.e. applications which do not violate the subformula property.

# 4 Modifications in the implementation

In his MSc. dissertation [Dia02], Wagner Dias presented a framework for tableaux methods. One of the classes in this frammework is the Formula class, which represents a formula. In the system implemented based on this framework, a formula object is constructed by parsing a string using a function called parse. This function does not identify when two subformulas of a formula are the same and it always creates new instances of any formula, either atomic or composite, that appears during the parse.

For instance, if the string is:

$$A \rightarrow (B \vee A)$$

the parse function creates 5 formulas (5 instances of the Formula class), one for each subformula of the main formula:

[2]In other words, for each analytic tableau proof of size $n$, there is a KE-Tableau proof with size polynomial in $n$. But there is at least one proof in KE-Tableau of size $n$ whose corresponding proof in Analytic Tableaux has size superpolynomial in $n$.

**Disjunction Rules**

$$\frac{\begin{array}{l} T\,A \vee B \\ F\,A \end{array}}{T\,B}\ (\mathrm{E}T \vee 1) \qquad \frac{\begin{array}{l} T\,A \vee B \\ F\,B \end{array}}{T\,A}\ (\mathrm{E}T \vee 2) \qquad \frac{F\,A \vee B}{\begin{array}{l} F\,A \\ F\,B \end{array}}\ (\mathrm{E}F\vee)$$

**Conjunction Rules**

$$\frac{\begin{array}{l} F\,A \wedge B \\ T\,A \end{array}}{F\,B}\ (\mathrm{E}F \wedge 1) \qquad \frac{\begin{array}{l} F\,A \wedge B \\ T\,B \end{array}}{F\,A}\ (\mathrm{E}F \wedge 2) \qquad \frac{T\,A \wedge B}{\begin{array}{l} T\,A \\ T\,B \end{array}}\ (\mathrm{E}T\wedge)$$

**Implication Rules**

$$\frac{\begin{array}{l} T\,A \to B \\ T\,A \end{array}}{T\,B}\ (\mathrm{E}T \to 1) \qquad \frac{\begin{array}{l} T\,A \to B \\ F\,B \end{array}}{F\,A}\ (\mathrm{E}F \to 2) \qquad \frac{F\,A \to B}{\begin{array}{l} T\,A \\ F\,B \end{array}}\ (\mathrm{E}F \to)$$

**Negation Rules**

$$\frac{T\,\neg A}{F\,A}\ (\mathrm{E}T\neg) \qquad \frac{F\,\neg A}{T\,A}\ (\mathrm{E}F\neg)$$

**Principle of Bivalence**

$$\frac{}{T\,A \mid F\,A}\ (\mathrm{PB})$$

Figure 2: KE tableau expansion rules

1. $A$
2. $B$
3. $A$
4. $B \wedge A$
5. $A \rightarrow (B \vee A)$

But since the third formula is equal to the first, both could be represented by two pointers to the same atom.

We modified Wagner Dias's system so that we have two vectors of formulas: one for atomic formulas and other for composite formulas. In this representation, for the formula above we would have:

| index | atomic formula |
|---|---|
| 0 | $A$ |
| 1 | $B$ |

| index | composite formula | internal representation |
|---|---|---|
| 2 | $B \vee A$ | $1 \vee 0$ |
| 3 | $A \rightarrow (B \vee A)$ | $0 \rightarrow 2$ |

In Wagner Dias's system, the prove program receives a file containing a set of signed formulas and builds a tableau for this set using one of the methods (analytic, KE or KES3) present in the framework. As a result, the program shows the total number of nodes of the tableau, the total number of formulae and the elapsed time (in seconds) neccessary for the construction of the tableau. Optionally, it also shows a text representation of the tableau proof. For instance,

```
prove -m analytic -f gamma1.prove
```

builds an analytic tableau proof for the $\Gamma_1$ set of formulas. And

```
prove -m ke -v -f gamma2.prove
```

builds and shows (-v option) a KE tableau proof for the $\Gamma_2$ set of formulas.

In both cases above, the readProve function is used to call parse on the set of formulas[3]. The readProve function receives as input a sequence of strings such as the one below for the $\Gamma_1$ set of formulas:

```
T(a1|b1)
T(a1->(a2|b2))
T(b1->(a2|b2))
F(a2|b2)
```

and returns true if all formulas are well-formed and false otherwise. Actually, it receives two parameters: the string with the formulas and a pointer to a vector of signed formulas. If the string contains only well-formed formulas, the vector of signed formulas will receive, for each line in the string, an object of the SignedFormula class. This object will be the result of calling the constructor of SignedFormula with two parameters:

1. the sign, which is one of the two constants of the SignedFormula class: SignedFormula::S_T or SignedFormula::S_F depending, of course, on the first character of the line and
2. the Formula object which is the result of applying parse to the characters on the line after the sign.

The code for this function is shown below:

[3]There is also a readCNF function which reads formulas in conjunctive normal form, but we have not used it.

```
bool readProve(const string& file, vector<SignedFormula *>& v)
{
  string s_fml;
  char sign;
  Formula *p_fml;

  ifstream in(file.c_str());

  in >> sign;
  in >> s_fml;
  while (!in.eof()) {
    p_fml = parse(s_fml);
    if (p_fml) {
      v.push_back(new SignedFormula(
sign=='T'?SignedFormula::S_T:SignedFormula::S_F,
p_fml));
    }
    else {
      in.close();
      return false;
    }
    in >> sign;
    in >> s_fml;
  }
  in.close();
  return true;
}
```

The call to readProve or readCNF functions depends on the extension of the file passed as a parameter to the prove program after the -f option. If the file extension is '.cnf' it calls readCNF, whilst if the file extension is '.prove', it calls readProve. Below we show the lines that declare the vector of signed formulas which receives the SignedFormula objects and, after that, call readCNF or readProve:

```
  vector<SignedFormula *> v;

  bool read_ok;
  if (cnf)
    read_ok = readCNF(file, v);
  else
    read_ok = readProve(file, v);
```

Then, in the main function a Tableau object is declared:

```
  Tableau *tab;
```

And this object is initialized with an object of the class that implements the specific tableau method chosen in the parameters passed to the prove program. The constructor of this class receives as parameter the vector containing the set of formulas. For, instance, for the '-m analytic' option, the program would initialize the Tableau object in the following way:

```
  if (method == "analytic") {
    tab = new AnalyticTableau("1", v);
    ((AnalyticTableau *) tab)->setStrategy(new AnalyticStrategy());
  }
```

Notice that in the last line a strategy is set for the Tableau object. The choice of strategy also depends on the parameters given to the prove program.

Finally, the line

```
bool closed = tab->close();
```

calls a method that tries to close the tableau. If it is successful, the method returns true. Otherwise, it returns false.

Following is the new version of the readProve function. It basically uses two vectors to store the atomic formulas and the composite formulas.

```
bool readProve(const string& file, vector<SignedFormula *>& v)
{
  string s_fml;
  char sign;
  Formula *p_fml;
  vector <Formula *> atomicFormulas, compoundFormulas;

  ifstream in(file.c_str());

  in >> sign;
  in >> s_fml;

  while (!in.eof()) {

    p_fml = parse2(s_fml, &atomicFormulas, &compoundFormulas);
    if (p_fml) {
      v.push_back(new SignedFormula(
sign=='T'?SignedFormula::S_T:SignedFormula::S_F,
p_fml));
    }
    else {
      in.close();
      return false;
    }
    in >> sign;
    in >> s_fml;
  }
  in.close();

  return true;
}
```

## 4.1 Modifications in the Formula class

Actually only one modification was necessary in the Formula class. The other modifications were made in functions that used instances of this classe. As one can see below, the destructor of the Formula class in Wagners system deallocates memory of all its subformulas. Now we cannot do this anymore because the subformulas of a formula can be subformulas of other formulas.

```
Formula::~Formula()
{

// removed because it is not adequate anymore
/* if (left)
delete left;
if (right)
delete right;
for (unsigned int i = 0; i < fmls.size(); i++)
delete fmls[i];
*/
}
```

These are new functions needed for the new parse:

```
// returns a previously allocated pointer to a formula (an element of the "formulas"
// vector) if there is a formula equal to "aFormula" in this vector. Otherwise,
// returns NULL.
Formula *includes (vector <Formula *> formulas, Formula * aFormula){

        for (unsigned int i = 0; i!=formulas.size(); i++){
                if (equalFormula (aFormula, formulas[i]))
                        return formulas[i];
        }

        return NULL;
};

// Insert a formula in the vector of formulas used by parse
// only if the formula is not there
void insert (vector <Formula *>  *formulas, Formula *aux,
                Formula **pFormula){


        // if the formula "aux" has not already been parsed
        Formula *aFormula = includes (*formulas, aux);
        if (aFormula == NULL){
        // returns it (as pFormula) and inserts it into the
        // vector of formulas
        (*pFormula) = aux;
        formulas->push_back(*pFormula);
        } else {
        // otherwise returns the previously allocated
        // formula "aFormula" and deletes the recently
        // created formula "aux".
        (*pFormula) = aFormula;

        delete aux;
        };

}
```

And below is the new parse function. It is named parse2. It receives three parameters – bsides the string from which the formula is parsed, it receives two pointers to the two vectors of atomic formulas and compound formulas already parsed in previous parses of the same tableau. This is necesseary so that for each tableau each formula be created only once.

```
// Parse a formula from a string. The formula (and its subformulas
// except atoms) must be enclosed in parenthesis. Returns a newly
// allocated pointer to the formula represented by the string or a
// null pointer if the string is not a valid formula.

// Receives a vector of atomic formulas and a vector of compound
// formulas of previous parses (if neccessary).
Formula *parse2(const string& s, vector <Formula *> *atomicFormulas,
                vector<Formula *> *compoundFormulas)
{

Formula *retval;
unsigned int i;
stack<parsed_item> S;
```

```
      for (i = 0; i < s.size(); ) {
   parsed_item item;
if (s[i] == ' ')
i++;
else
if (s[i] == '(') {
      item.type = PARSE_OPEN;
  S.push(item);
  i++;
} else
if (isalnum(s[i])) {
   string a;
   while (i < s.size() && (isalnum(s[i]) || s[i] == '_' || s[i] == ',')) {
a += s[i];
i++;
   }

item.type = PARSE_FORM;
if (!S.empty() && S.top().type == PARSE_OPER && S.top().op == Formula::NOT) {

// insert a negated atomic formula in the list of formulas
                 // (if it is a new one)

// creates a new atom only if necessary
Formula * auxAtom1 = new Formula(a),  *auxAtom2;
insert (atomicFormulas, auxAtom1, &auxAtom2);
Formula * aux = new Formula(Formula::NOT, auxAtom2);
insert (compoundFormulas, aux, &(item.formula));

S.pop();
}else {
// insert an atomic formula in the list of formulas
                 // (if it is a new one)
Formula * aux = new Formula(a);
insert (atomicFormulas, aux, &(item.formula));
}
S.push(item);
}else
if (s[i] == '!') {
item.type = PARSE_OPER;
item.op = Formula::NOT;
S.push(item);
i++;
} else
if (s[i] == '&') {
item.type = PARSE_OPER;
item.op = Formula::AND;
S.push(item);
i++;
} else
if (s[i] == '|') {
item.type = PARSE_OPER;
item.op = Formula::OR;
S.push(item);
```

```
i++;
}
else
if (i+1 < s.size() && s[i] == '-' && s[i+1] == '>') {
item.type = PARSE_OPER;
item.op = Formula::IMPLIES;
S.push(item);
i = i + 2;
}
else
if (s[i] == ')') {
Formula *result;

assert(!S.empty() && S.top().type == PARSE_FORM);

vector<Formula *> fmls;
fmls.push_back(S.top().formula);
S.pop();

assert(!S.empty() &&
(S.top().type == PARSE_OPEN || S.top().type == PARSE_OPER));

Formula::opType op = S.top().op;
while (S.top().type == PARSE_OPER) {
assert(op == S.top().op);
S.pop();

assert(!S.empty() && S.top().type == PARSE_FORM);

// puts a formula in fmls
fmls.insert(fmls.begin(), S.top().formula);
S.pop();
}

// removes opening parenthesis from the stack
assert(!S.empty() && S.top().type == PARSE_OPEN);

S.pop();

if (fmls.size() > 2) {
assert (op == Formula::AND || op == Formula::OR);
if (op == Formula::AND)
op = Formula::ANDN;
else
op = Formula::ORN;

            Formula *aux = new Formula (op, fmls);
                  insert (compoundFormulas, aux, &result);

}else if (fmls.size() == 2){

Formula *aux = new Formula(op, fmls[0], fmls[1]);

// insert a formula in the list of formulas
                // (if it is a new one)
```

```
insert (compoundFormulas, aux, &result);

}
else // fmls.size() == 1
result = fmls[0];

item.type = PARSE_FORM;
if (!S.empty() && S.top().type == PARSE_OPER &&
S.top().op == Formula::NOT) {
// insert a formula in the list of formulas
                // (if it is a new one)
Formula *aux = new Formula(Formula::NOT, result);
insert (compoundFormulas, aux, &(item.formula));
S.pop();
} else
item.formula = result;
S.push(item);
i++;
}
}

assert(!S.empty() && S.top().type == PARSE_FORM);
retval = S.top().formula;
S.pop();
assert(S.empty());

return retval;
}
```

We kept a parse function that calls parse2 passing two vectors that are not further used. It makes use of the equalVectors function right below it.

```
// Calls parse2 passing two vectors. Used when you do not want
// to take into account the atomic formulas and the compoound
// formulas.
Formula *parse(const string& s)
{

        vector<Formula *> atomicFormulas, compoundFormulas;

        return parse2 (s, &atomicFormulas, &compoundFormulas);


}
```

In order to check if a formula was equal to another formula already parsed we had to implement teh following function that checks if the content of a formula object is equal to anothers content.

```
// Returns true if a  formula is syntactically equal to another formula
bool equalFormula (Formula* aFormula, Formula *anotherFormula){

if (aFormula->op != anotherFormula->op){
return false;
}
```

```
else {

switch (aFormula->op) {
case Formula::ATOM:
{

if (aFormula->atom == anotherFormula->atom)
return true;
else
return false;
}
break;
case Formula::NOT:
{
return equalFormula (aFormula->right, anotherFormula->right);
}
break;
case Formula::IMPLIES:
{
return (equalFormula (aFormula->right, anotherFormula->right) &&
equalFormula (aFormula->left, anotherFormula->left));
}
break;
case Formula::OR: case Formula::AND:
{
return ( (equalFormula (aFormula->right, anotherFormula->right)
                &&
  equalFormula (aFormula->left, anotherFormula->left)));
}
break;
case Formula::ORN: case Formula::ANDN:
{

if (equalVectors (aFormula->fmls, anotherFormula->fmls))
return true;
else  return false;

}
}

};

return false;

}


// returns true if the vectors contain the same formulas
// (according to equalFormula) in the same order
bool equalVectors (vector<Formula *> v1, vector<Formula *> v2){

  if (v1.size() != v2.size())
return false;
  else {
  for (unsigned int i = 0; i!=v1.size(); i++){
```

```
       if (!(equalFormula (v1[i], v2[i]) )){
       return false;
}
   }

  }

  return true;
};
```

## 4.2 Modifications in other classes

Having modified the parsing of formulas, we had to change more three classes (Tableau, TableauStrategy and KETableau) so that we could use the information about atomic and composite formulas of a tableaux.

First, we had to make a modification in the format of rules that are used by tableau objects and tableau strategy objects. Previously, these rules received only two parameters: the vectors of signed formulas that are, respectively, input to and output from a rule. Now we have to receive also the vector of all signed formulas in the tableau as well as the vectors of atomic and compound formulas.

```
//////////////////////////////////////////////////////////////////////////////
// A rule is a pointer to a function returning bool and having the
// input and output formulas as parameters.
//////////////////////////////////////////////////////////////////////////////

typedef bool (*Rule)(const vector<SignedFormula *>&, vector<SignedFormula *>&,
                                vector<SignedFormula *>&,
                           vector<Formula *>&, vector<Formula *>&);
```

We had to make some changes in the TableauStrategy class, always following this pattern: adding the vector of all signed formulas in the tableau as well as the vectors of atomic and compound formulas.

```
class TableauStrategy
{
 public:
  TableauStrategy();
  virtual ~TableauStrategy();

  // initializes the object with the items of a tableau. Returns true
  // if the tableau is already close.  This method also uses the
  // Floyd-Warshall algorithm to calculate the minimum distance
  // between each pair of atoms.
  virtual bool init(const string& tableau_id,
    vector<SignedFormula *> *items,
    vector<SignedFormula *> *alphas,
    vector<SignedFormula *> *betas,
    vector<SignedFormula *> *lits,
    vector<Formula *> *atomicFormulas,
    vector<Formula *> *compoundFormulas
    );

  (...)

     // Atomic formulae of the tableau.
  vector<Formula *> *_atomicFormulas;
```

```
    // Compound formulae of the tableau.
  vector<Formula *> *_compoundFormulas;


  (...)

}
```

And it was neccessary to make modifications in the applyRule method that calls rules, since we changed the format of rules:

```
bool Tableau::applyRule(unsigned int index,
const vector<SignedFormula *>& in,
vector<SignedFormula *>& out,
                        vector<SignedFormula *>& items,
vector<Formula *>& atomicFormulas,
                        vector<Formula *>& compoundFormulas)
{
  if (index < _rules.size())
    return (*_rules[index])(in, out, items, atomicFormulas, compoundFormulas);
  return false;
}
```

The implementation of all rules had to be modified because no new formulas or signed formulas should be created by them. In order to achieve this objective, we had to implement four functions:

- aSignedFormula – returns a pointer to a new signed formula if there is no signed formula in items (with the same sign and formula pointer) or to an existing formula.
- pushBackIfNew – very similar to the previous function, it pushes a new signed formula in items only if it is not already there.
- aSignedFormulaIndex is the same as aSignedFormula only that it returns an index to the formula in the items vector (if it is there) or -1 (in case it is not there).
- aFormulaIndex – returns an index of a formula in a vector of formulas, or -1 if the vector does not include the formula.

The code for these functions is presented below:

```
SignedFormula *aSignedFormula(SignedFormula::Sign sign, Formula *fml,
        vector<SignedFormula *>& items,
        vector<Formula *>& atomicFormulas, vector<Formula *>& compoundFormulas){

        // verifies if the Signed Formula is in items. If it is, returns it. Otherwise,
        // creates it.

        for (unsigned int i=0; i< items.size(); i++){
                if ( ( (items[i])->sign  == sign   ) &&
                     ( (items[i])->formula == fml  ) ) {
                     return items[i];
                };
        };

        return new SignedFormula (sign, fml);
}

void pushBackIfNew (vector <SignedFormula *>& aVector, SignedFormula::Sign sign, Formula *fml,
                vector<SignedFormula *>& items,
```

```
            vector<Formula *>& atomicFormulas, vector<Formula *>& compoundFormulas){

        // verifies if the Signed Formula is in items. If it is, returns it. Otherwise,
        // creates it.

        for (unsigned int i=0; i< items.size(); i++){
                if ( ( (items[i])->sign  == sign   ) &&
                     ( (items[i])->formula == fml  ) ) {
                     return;
                };
        };

        aVector.push_back(new SignedFormula (sign, fml));;
}


int aSignedFormulaIndex(SignedFormula::Sign sign, Formula *fml,
        vector<SignedFormula *>& items,
        vector<Formula *>& atomicFormulas, vector<Formula *>& compoundFormulas){

        // verifies if the Signed Formula is in items. If it is, returns its index. Otherwise,
        // returns -1.

        for (unsigned int i=0; i< items.size(); i++){
                if ( ( (items[i])->sign  == sign   ) &&
                     ( (items[i])->formula == fml  ) ) {
                     return i;
                };
        };

        return -1;
}

int aFormulaIndex(Formula *fml, vector<Formula *>& formulas){

        // verifies if the formula is in vector formulas. If it is, returns its index. Otherwise,
        // returns -1.

        for (unsigned int i=0; i< formulas.size(); i++){
                if  ( formulas[i] == fml)  {
                     return i;
                };
        };

        return -1;
}
```

Below is an example of a rule whose implementation was changed:

```
bool KE_alpha_E_NOT_OR(const vector<SignedFormula *>& in,
    vector<SignedFormula *>& out,
    vector<SignedFormula *>& items,
                    vector<Formula *>& atomicFormulas, vector<Formula *>& compoundFormulas)
{
  if (! (in.size() == 1 && in[0] && in[0]->sign == SignedFormula::S_F &&
 in[0]->formula && in[0]->formula->op == Formula::OR)) return false;
```

```
  pushBackIfNew(out,SignedFormula::S_F,
  in[0]->formula->left,
                              items, atomicFormulas, compoundFormulas);
  pushBackIfNew(out, SignedFormula::S_F,
  in[0]->formula->right,
                                items, atomicFormulas, compoundFormulas);

  return true;
}
```

# 5 Families of formulas

In order to make comparative tests for the tableaux methods implemented in his dissertation, Dias [Dia02] considered the following sets of formulas that are well known as difficult.

## 5.1 $\Gamma$ formulas

The $\Gamma$ family of formulas [CS00] has the following form:

$$\Gamma_i = \left\{ \begin{array}{c} p_1 \to (p_2 \vee q_2) \\ q_1 \to (p_2 \vee q_2) \\ \vdots \\ p_i \to (p_{i+1} \vee q_{i+1}) \\ q_i \to (p_{i+1} \vee q_{i+1}) \end{array} \right.$$

The $\Gamma_n$ family has $2n$ formulas and $2n$ propositional variables. The sequent that one wants to prove is:

$$p_1 \vee q_1, \Gamma_n \vdash p_{n+1} \vee q_{n+1}$$

This family has cut-free proofs of size exponential and non-exponential in $n$.

## 5.2 $H$ formulas

The $H$ formulas, or fat formulas, were defined in [DAg92] in order to prove that Smullyan's analytic tableaux did not polinomially simulate truth-tables. A $H$ formula is constructed in the following way: let $p_1, \ldots, p_n$ be propositional variables and consider all possible disjunctions containing either $l_i$ or $\overline{l_i}$, where $l_i = p_i$ and $\overline{l_i} = \neg p_i$, and $i = 1, \ldots, n$. The problem consists in showing that the conjunction of these disjunctions is unsatisfiable. The size of the problem is the number of variables $n$. The initial number of formulas is one but the size of this formula is of order $2^n$.

## 5.3 Statman formulas

The Statman formulas [Sta78] have the following form:

$$\begin{array}{c} p_0 \vee q_0 \\ ((p_0 \vee q_0) \to p_1) \vee ((p_0 \vee q_0) \to q_1) \\ ((p_0 \vee q_0) \wedge (p_1 \vee q_1) \to p_2) \vee ((p_0 \vee q_0) \wedge (p_1 \vee q_1) \to q_2) \\ \vdots \\ ((p_0 \vee q_0) \wedge (p_1 \vee q_1) \wedge \ldots \wedge (p_{i-1} \vee q_{i-1}) \to p_i) \vee ((p_0 \vee q_0) \wedge (p_1 \vee q_1) \wedge \ldots \wedge (p_{i-1} \vee q_{i-1}) \to q_i) \end{array}$$

The problem is to decide if $p_i \wedge q_i$ can be deduced from the set of formulas above. This family admits short proofs with the use of the cut rule, but only exponential proof without cut.

### 5.4 PHP formulas

The formulas in the PHP (Pigeon Hole Principle) familiy [CS00] describe the problem of placing $n+1$ letters in $n$ pigeon holes. According to the Pigeon Hole Principle, there is always one pigeon hole which will receive at least two letters. The family has the following form:

$$
\begin{array}{cc}
(p_{11} \vee p_{12} \vee \ldots \vee p_{1\ n-1}) & \wedge \\
(p_{21} \vee p_{22} \vee \ldots \vee p_{2\ n-1}) & \wedge \\
\vdots & \wedge \\
(p_{n1} \vee p_{n2} \vee \ldots \vee p_{n\ n-1}) & \rightarrow \\
((p_{11} \wedge p_{21}) \vee (p_{11} \wedge p_{31}) \vee \ldots \vee (p_{11} \wedge p_{n1}) & \wedge \\
((p_{21} \wedge p_{31}) \vee (p_{21} \wedge p_{41}) \vee \ldots \vee (p_{21} \wedge p_{n1}) & \wedge \\
\vdots & \wedge \\
((p_{n-1\ 1} \wedge p_{n1})) & \vee \\
\vdots & \\
((p_{1n} \wedge p_{2n}) \vee (p_{1n} \wedge p_{3n}) \vee \ldots \vee (p_{1n} \wedge p_{n\ n}) & \wedge \\
((p_{2n} \wedge p_{3n}) \vee (p_{2n} \wedge p_{4n}) \vee \ldots \vee (p_{2n} \wedge p_{n\ n}) & \wedge \\
\vdots & \wedge \\
((p_{n-1\ n} \wedge p_{n\ n})) &
\end{array}
$$

where $p_{ij}$ means that the letter $i$ is in pigeon hole $j$. The problem consists in proving that the formula above is valid. The problem with size $n$ has $n*(n-1)$ propositional variables and the formula's size is of order $n^3$. Cut-free proofs of this problem are exponential. This problem causes a lot of branching and seems to be exponential, although there is a proof by Buss that is polynomial.

## 6 Test results

We have tested both implementations with some instances of the families of formulas presented in section [?] in a personal computer. One can notice that although the number of formulas in the closed tableaus decreased a little, the number of nodes remained the same and the time spent for the closing of each tableau did not always decrease. We believe this happened because there is an overhead associated with the sharing of subformulas. Below we present the results.

### 6.1 $\Gamma$ formulas

| | Changed version | | | Original version | | |
|---|---|---|---|---|---|---|
| Instance | # nodes | # formulas | Time (s) | # nodes | # formulas | Time (s) |
| $\Gamma_1$ | 3 | 13 | 0.000363 | 3 | 13 | 0.000351 |
| $\Gamma_2$ | 13 | 41 | 0.002388 | 13 | 42 | 0.002353 |
| $\Gamma_3$ | 37 | 112 | 0.011127 | 37 | 117 | 0.010763 |
| $\Gamma_4$ | 109 | 321 | 0.086773 | 109 | 338 | 0.047102 |
| $\Gamma_5$ | 325 | 944 | 0.283089 | 325 | 997 | 0.193968 |
| $\Gamma_6$ | 973 | 2809 | 0.955900 | 973 | 2970 | 0.927524 |
| $\Gamma_7$ | 2917 | 8400 | 3.635644 | 2917 | 8885 | 3.410045 |
| $\Gamma_8$ | 8749 | 25169 | 13.903300 | 8749 | 26626 | 12.979512 |

### 6.2 $H$ formulas

| | Changed version | | | Original version | | |
|---|---|---|---|---|---|---|
| Instance | # nodes | # formulas | Time (s) | # nodes | # formulas | Time (s) |
| $H_1$ | 1 | 4 | 0.000040 | 1 | 4 | 0.000048 |
| $H_2$ | 3 | 15 | 0.000938 | 3 | 17 | 0.000329 |
| $H_3$ | 21 | 99 | 0.004384 | 21 | 103 | 0.004664 |
| $H_4$ | 105 | 669 | 0.094125 | 105 | 677 | 0.100072 |
| $H_5$ | 465 | 4761 | 2.574431 | 465 | 4777 | 2.709676 |

### 6.3 Statman formulas

| | Changed version | | | Original version | | |
|---|---|---|---|---|---|---|
| Instance | # nodes | # formulas | Time (s) | # nodes | # formulas | Time (s) |
| $\text{Statman}_1$ | 1 | 5 | 0.000063 | 1 | 5 | 0.000069 |
| $\text{Statman}_2$ | 7 | 30 | 0.000901 | 7 | 30 | 0.000876 |
| $\text{Statman}_3$ | 31 | 124 | 0.008043 | 31 | 124 | 0.008224 |
| $\text{Statman}_4$ | 207 | 697 | 0.139583 | 207 | 711 | 0.098986 |
| $\text{Statman}_5$ | 1079 | 3538 | 1.125550 | 1079 | 3696 | 1.176937 |
| $\text{Statman}_6$ | 5391 | 18371 | 10.263416 | 5391 | 19539 | 10.126263 |

### 6.4 PHP formulas

| | Changed version | | | Original version | | |
|---|---|---|---|---|---|---|
| Instance | # nodes | # formulas | Time (s) | # nodes | # formulas | Time (s) |
| $\text{PHP}_1$ | 1 | 6 | 0.000072 | 1 | 6 | 0.000088 |
| $\text{PHP}_2$ | 3 | 26 | 0.000674 | 3 | 29 | 0.000694 |
| $\text{PHP}_3$ | 61 | 251 | 0.045288 | 61 | 261 | 0.046799 |
| $\text{PHP}_4$ | 483 | 2159 | 1.917854 | 483 | 2219 | 2.385304 |

## 7 Conclusion

In his MSc. Dissertation [Dia02], Wagner Dias presented an object-oriented framework for tableaux methods. And he also gave an implementation of this framework in the C++ programming language. His implementation covered three tableaux methods: Smullyan's Analytic Tableaux, D'Agostino's KE Tableaux and Finger and Wasserman's KE-$S_3$ Tableaux. Our problem was to make some modifications in his implementation so that KE Tableau proofs became shorter. With this in mind, we first had to make changes in the representation of formulas, because the original implementation did not take advantage of the fact that some formulas may share subformulas. Then we had to modify the Formula class, as well as the classes Tableau, TableauStrategy and KETableau. In this paper we presented the major modifications we have made.

We have run tests with both implementations and noticed that, although the number of formulas in the closed tableaus decreased a little, the number of nodes remained the same and the time spent for the closing of each tableau did not always decrease, because there is an overhead associated with the sharing of subformulas.

### 7.1 Future works

First we will try to reduce the number of nodes of some closed KE Tableaus by allowing closing not only with atomic formulas, but also with composite formulas (e.g. $T\,A \vee B$ and $F\,A \vee B$). This will be relatively easy to implement now that we have changed the representation of formulas. Let us see if this diminishes or increases the time spent for closing, since such strategy will result in more formulas to look at when searching for a closing case.

Next, by also making use of the new representation of formulas, we shall try to apply some KE beta rules faster. For instance, if $T\,X \rightarrow B$ and $T\,X$, then just by noticing that $X$ in both signed formulas is the same formula, we can arrive at $T\,B$.

And finally, in a later stage, we intend to use aspect-oriented programming techniques for the development of a more general framework for tableaux methods. Wagner Dias's framework is limited to classical logic. Why not have a framework also for implementing tableaux methods for substructural logics or fuzzy logic, for instance? Maybe aspects, by giving us another dimension, can help us with that job.

# A A personal evaluation on the experience of rewriting code

In Software Engineering books there are many listings of the advantages and disadvantages of rewriting code. Here I will give a personal evaluation based on my experience with Wagner Dias's C++ implementation.

It is obvious that the first thing one must do when he wants to modify somebody else's system is to understand what has been done. It is not difficult to understand what Wagner has done because his code is reasonably well written and well commented. That is, classes, instance variables and methods have adequate names, and there are some comments for each major class or method. Besides that, his dissertation describing the framework he used helped a lot.

But of course there were problems:

- There were some errors caused by something obvious that I did not perceive until it happened. We all known that a change in one place may always cause an unantecipated result in other part of the system, causing an error. Sometimes this 'unanticipated' result is not so surprising – the problem is that we were not careful enough. Computer programming is an activity that stimulates lack of discipline, says Valdemar Setzer. For instance, Wagner's destructor for the Formula class recursively deallocated memory for all the subformulas of a formula. Since in our version we could be sharing subformulas with other formulas, the destructor had to be modified (actually it does nothing now). But I only noticed that when the system crashed ...
- I did not use any debugging environment. If the system were written in Smalltalk, for example, it would be much easier to discover the contents of each data structure during execution, because Smalltalk has an excellent debugging environment.
- There are some decisions that may help in one aspect but be prejudicial in others. For example, in a Tableau object, all signed formulas of the tableau are stored in a vector called _items. But when this tableau is the child of another tableau, only the formulas that are new to this child are stored in _items. This saves memory but makes it more difficult to discover a closing case and to check if a signed formula that is going to be added to the tableau is already in its list of signed formulas. But when I tried to change _items so that it would include pointers to all signed formulas of the tableau up to that child, I discovered that the toString() method used _items to print a tableau proof, so I could not change this without some major modifications in other parts of the system.
- I did not have any communication with the writer of the code.

Whenever a problem occured I had to investigate the code in order to discover where and why it happened. I did not think it was neccessary to study all the code at once, so I studied only the parts I would modify. And actually there are still some parts of the system whose working I still do not understand. However, there were some parts of the code that at first I thought it would not be necessary to understand but that ended up being relevant for my modifications.

In conclusion, it is a very enriching experience because I was really obliged to understand much of Wagner Dias's system.

## References


[Bet59] E. W. Beth. The Foundations of Mathematics, North Holland, 1959.

[CS00] A. Carbone and S. Semmes. *A graphical apology symmetry and implicitness.* Oxford mathematical Monographs, 2000.

[Dia02] Wagner Dias. *Implementações de Tableaux para Raciocínio por Aproximações.* Dissertação de Mestrado, Departamento de Ciência da Computação, Instituto de Matemática e Estatística, Universidade de São Paulo, 2002.

[DAg92] Marcello DAgostino. *Are tableaux improvement on truth-tables?* Journal of Logic, Language and Information, 1992. Kluwer Academic Publishers, 1999.

[DAg99] Marcello DAgostino. *Tableau Methods for Classical Propositional Logic.* In: Handbook of Tableau Methods, Marcello DAgostino et al. (eds), 45-123. Kluwer Academic Publishers, 1999.

[Fit99] Melvin Fitting. *Introduction.* In: Handbook of Tableau Methods, Marcello DAgostino et al. (eds), 1-43. Kluwer Academic Publishers, 1999.

[Gab96] Dov Gabbay. *Labelled Deductive Systems, Volume 1 - Foundations.* Oxford University Press, 1996.

[Sta78] R. Statman. *Bounds for proof-search and speed-up in the predicate calculus.* Annals of Mathematical Logic, 1978. Oxford University Press, 1996.

[Smu68] Raymond M. Smullyan. *First-Order Logic.* Springer-Verlag, 1968.

[Hin55] J. Hintikka. Form and content in quantification theory. *Acta Philosophica Fennica.* **8**, 7-55, 1955.

[Gen35] G. Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift 39,* 176-210, 405-431, 1935. English translation, 'Investigations into logical deduction', in [Sza69].

[Mor01] Mortari, Cezar. Introdução à Lógica. Editora da UNESP, So Paulo, 2001.

[Sza69] M. E. Szabo, Ed. *The Collected Papers of Gehrard Gentzen.* North-Holland, Amsterdam, 1969.