Using Aspect-Oriented Programming in the Development of a Multi-Strategy Theorem Prover

Adolfo G. S. Seca Neto[1] **and Marcelo Finger**[1]

[1] Departamento de Ciência da Computação,
Instituto de Matemática e Estatística – Universidade de São Paulo (USP),
Rua do Matão, 1010, São Paulo – SP – Brazil – 05.315-970,
e-mail: [adolfo, mfinger]@ime.usp.br

Abstract. *When a computer program is written to implement a nondeterministic algorithm, it must have a strategy for choosing the next step that is going to be performed. Automated Theorem Provers (ATPs) usually implement nondeterministic algorithms, therefore the representation of strategies is a very important part of their design. A multi-strategy theorem prover is an ATP where we can vary strategies without modifying the implementation. In this paper we present some remarks about the development of a multi-strategy prover implemented using Aspect-Oriented Programming.*

Resumo. *Quando um programa de computador é escrito para implementar um algoritmo não-determinístico, ele deve possuir uma estratégia para escolher o próximo passo a ser executado. Provadores Automáticos de Teoremas (PAT's) geralmente implementam algoritmos não-determinísticos, portanto a representação de estratégias é uma parte muito importante do seu projeto. Um provador de teoremas multi-estratégia é um PAT onde podemos variar estratégias sem modificar a implementação. Neste artigo apresentamos algumas observações sobre o desenvolvimento de um provador multi-estratégia implementado com programação orientado a aspectos.*

1. Introduction

An algorithm is a sequence of computational steps that takes a value (or set of values) as input and produces a value (or set of values) as output [Cormen et al. 2001]. A nondeterministic algorithm is an algorithm that is allowed, at certain times, to choose between two or more possible steps. Nondeterministic algorithms compute the same class of functions as deterministic algorithms, but the complexity is usually lower. Nondeterministic algorithms are used in several areas such as automated theorem proving, term-rewriting systems, and protocol specification.

Every nondeterministic algorithm can be turned into a deterministic algorithm, possibly with exponential slow down. For instance, there are some problems for which there is a polynomial-time nondeterministic algorithm but no polynomial-time deterministic one is known. And an exponential-time deterministic algorithm is obtained by testing all possibilities of the polynomial-time nondeterministic algorithm. One of the most important open research problems in computer science nowadays is the "P=NP?" question. Informally speaking, the answer to this question corresponds to knowing if decision problems that can be solved by a polynomial-time nondeterministic algorithm can also be solved by some polynomial-time deterministic algorithm.

When a computer program is written to implement a nondeterministic algorithm, it must have a strategy for choosing the next step that is going to be performed. An interesting example of nondeterminism and the use of strategies is the method of tableaux. It is a formal proof procedure that has many variants and exists for several logics [Fitting 1999]. It is a refutational procedure. That is, in order to prove that a formula X is valid we try to show that it is not valid. Having this in mind, we apply a procedure for inferring the logical consequences of the formulas present in the tableaux. This procedure applies a strategy for choosing the next expansion rule to be applied amongst possibly many applicable rules. A tree is generated by this procedure and if all branches of the tree close (a branch is closed when we find a contradiction) then we have a proof of X. Otherwise, if we apply all possible rules and at least one branch of tree remains open, we have a refutation of X.

Automated theorem provers were one of the first applications of computers and still have many applications such as hardware and software verification. Their history is almost as old as that of computing; the first provers were implemented almost 50 years ago. Most automated theorem provers nowadays are based either on the resolution principle [Robinson 1965] or on the DPLL procedure [Davis et al. 1962], but tableau methods have also been found to be a convenient formalism for automating deduction in various non-standard logics [Fariñas del Cerro et al. 2001] as well as in classical logic.

In our work we are investigating the construction of multi-strategy theorem provers (MSTPs). A MSTP is a theorem prover where we can vary the strategy without modifying the core of the implementation. A MSTP can be used for three purposes: educational, exploratory and adaptive. For educational purposes, it can be used to illustrate how the choice of a strategy can affect prover performance. As an exploratory tool, a MSTP can be used to test strategies and make comparisons between them. And we can also think of an adaptive multi-strategy theorem prover that changes the strategy used according to features of the problem presented to it.

Another objective is to investigate if proof strategies for tableau provers can be well modularized by using object-oriented and aspect-oriented programming. Aspect-Oriented Programming (AOP) or Aspect-Oriented Software Development (AOSD) is a new approach to software development that addresses limitations inherent in other approaches, including Object-Oriented Programming. The main motivation for the development of AOP was the alleged inability of object-oriented programming (OOP) and other current software development paradigms to fully support the *separation of concerns* principle [Kiczales et al. 2001]. According to AOP proponents, AOP solves some problems of OOP by allowing an adequate representation of the so-called *crosscutting concerns* [Soares and Borba 2002]. With AOP, code that implements crosscutting concerns, i.e. that implements specific functions that affect different parts of a system, and would be scattered and tangled in an OOP implementation, can be localized, increasing modularity. With this increase in modularity, one can achieve software that is more adaptable, maintainable and evolvable in the face of changing requirements.

The first step towards the construction of an multi-strategy theorem prover using AOP was the implementation of a single-strategy object-oriented version of a tableau prover [Neto 2003]. This prover was based on the KE Tableau System [D'Agostino and Mondadori 1994], developed by Marco Mondadori and Marcello

D'Agostino, for classical propositional logic. After this implementation, we posed ourselves the following questions:

- how can we use object-orientation and aspect-orientation to achieve modularity in the definition of proof strategies?
- how should we represent a proof strategy: as an object, an aspect or as an object whose behaviour is modified by aspects?
- can we represent features of proof strategies as advice methods?

The second step was to implement a multi-strategy theorem prover for classical propositional logic [Neto and Finger 2005] where aspects where used for profiling and strategy feature variation. Based on this experience, in this paper we will present our ideas about the construction of multi-strategy provers using AOP. In Section 2 we present the KE System and an example showing the use of strategies in that system. Section 3 presents some remarks on the design and implementation of a multi-strategy prover using aspect-orientation. Section 4 concludes and points to some future work.

2. The KE System

The KE System, a tableau method developed by Marco Mondadori and Marcello D'Agostino [D'Agostino and Mondadori 1994], was presented as an improvement, in the computational sense, over Analytic Tableaux [Smullyan 1968]. Here we discuss the version for classical propositional logic, a refutation system that is sound and complete.

We assume familiarilty with the syntax and semantics of propositional classical logic. See [Smullyan 1968] for an introduction. Let us see some conventions used throghout this paper. A *signed formula* is an expression $S\,X$ where S is called the *sign* and X is a propositional *formula*. The symbols $\mathtt{T}$ and $\mathtt{F}$, respectively representing the truth-values true and false, can be used as signs. The *conjugate* of a signed formula $\mathtt{T}\,A$ (or $\mathtt{F}\,A$) is $\mathtt{F}\,A$ (or $\mathtt{T}\,A$).

We define a *proof* in the KE System as a tree whose nodes are lists of signed formulas, here called *branch nodes*. The *root branch node* is the only branch node that does not have a parent branch node. All branch nodes can have two children: the *left branch node* child and the *right branch node* child. A *leaf branch node* is childless. A *branch* is a sequence of branch nodes starting at the root and finishing in a leaf branch node.

When we want to prove that the formulas $B_1, B_2, \ldots, B_n$ follow from $A_1, A_2, \ldots, A_m$, we start a tree with a root branch node containing the following sequence of formulas: $\mathtt{T}\,A_1, \mathtt{T}\,A_2, \ldots, \mathtt{T}\,A_m, \mathtt{F}\,B_1, \mathtt{F}\,B_2, \ldots, \mathtt{F}\,B_n$. That means we are trying to falsify the formula $(A_1 \wedge A_2 \wedge \ldots \wedge A_m) \rightarrow (B_1 \vee B_2 \vee \ldots \vee B_n)$. The set of expansion rules for the KE System is presented in Figure 1. For each branch node, we can use expansion rules that take as premises one or more signed formulas that already appear in the branch of this branch node and introduce one or more new signed formulas. These new signed formulas are logical consequences of the premises. The PB rule has no premiss and introduces two branch nodes as children of a given branch node.

The rules define what one can do, not what one must do. That is, at a given time during the construction of the tree one may have several rules that can be applied. Notice also that all rules are linear, except the PB rule, corresponding to the princi-

ple of bivalence. This rule is related to the Cut rule in Gentzen sequent presentations [Gentzen 1969].

$$\frac{T\,A \vee B \quad F\,A}{T\,B}\ (T\vee 1) \qquad \frac{T\,A \vee B \quad F\,B}{T\,A}\ (T\vee 2) \qquad \frac{F\,A \vee B}{F\,A \mid F\,B}\ (F\vee)$$

$$\frac{F\,A \wedge B \quad T\,A}{F\,B}\ (F\wedge 1) \qquad \frac{F\,A \wedge B \quad T\,B}{F\,A}\ (F\wedge 2) \qquad \frac{T\,A \wedge B}{T\,A \quad T\,B}\ (T\wedge)$$

$$\frac{T\,A \to B \quad T\,A}{T\,B}\ (T\to 1) \qquad \frac{T\,A \to B \quad F\,B}{F\,A}\ (F\to 2) \qquad \frac{F\,A \to B}{T\,A \mid F\,B}\ (F\to)$$

$$\frac{T\,\neg A}{F\,A}\ (T\neg) \qquad \frac{F\,\neg A}{T\,A}\ (F\neg) \qquad \frac{}{T\,A \mid F\,A}\ (\text{PB})$$

Figure 1. KE tableau expansion rules

A proof terminates when all branches of a tree are closed. It important to notice that closure is not a rule, but a definition. A branch is closed if, for some formula X, $\mathtt{T}\,X$ appears in some branch node and $\mathtt{F}\,X$ also appears in some branch node (possibly not the same) of the branch. That is, a branch is closed when we arrive at a contradiction. If we arrive at a contradicition in all branches of the generated tree, then the formula we were trying to falsify is valid (that is, $B_1, B_2, \ldots, B_n$ follow from $A_1, A_2, \ldots, A_m$). Otherwise, it is not valid.

The *size* of a tableau proof is defined as the sum of the sizes of all branch nodes of the proof tree generated by the use of expansion rules. The size of a branch node is the sum of the size of all its signed formulas. The size of a signed formula is the size of its formula. Finally, the size $s(A)$ of a formula A is defined as:

- $s(A) = 1$ if A is a propositional atom;
- $s(\neg A) = 1 + s(A)$, where A is a formula and
- $s(A \circ B) = 1 + s(A) + s(B)$, where $\circ$ is a binary connective, and A and B are formulas.

The *height* of the proof tree and the number of branch nodes in the tree are other important dimensions of a proof. These are defined as usually for trees.

Let us give an example of proof in the KE System (see the first proof in Figure 2) that will help to illustrate the use of strategies in tableaux. The formula below, called Γ_3, is a tautology:

$$((p_1 \vee q_1)\wedge$$
$$(p_1 \to (p_2 \vee q_2)) \wedge (q_1 \to (p_2 \vee q_2))\wedge$$
$$(p_2 \to (p_3 \vee q_3)) \wedge (q_2 \to (p_3 \vee q_3))\wedge$$
$$(p_3 \to (p_4 \vee q_4)) \wedge (q_3 \to (p_4 \vee q_4))) \to$$

$$(p_4 \vee q_4)$$

Suppose we want to prove this formula, representing it as the signed formulas 1-8 in Figure 2. First all linear rules are applied. This generates formulas 9-12. Then, one has to choose a formula to apply the PB rule. It is clever to choose a formula that can be used as an auxiliary premise with one of the five formulas (1-5) that were not yet used as main premises. If we first choose the left subformula of 2, the result is a proof with size 71 and 31 nodes. If we use a different strategy, do not expand formula 8 and choose the left subformula of 4 to apply the PB rule, the result is a proof with size 61 and 25 nodes, as can be seen in the second proof of Figure 2.

3. Design

Here we will describe some of our ideas about the design and implementation of a multi-strategy theorem prover using AOP. A simplified class diagram of the system is presented in Figure 3. The Prover class has a prove method that receives a Problem object and returns a Proof. A Problem object contains a list of signed formulas along with data structures that hold these formulas. These data structures (factories of formulas) go along because we are using the Flyweight design pattern [Gamma et al. 1994]. This pattern prevents the multiplication of objects representing formulas and signed formulas as well as makes it easier to implement rule choice and application. By using this pattern, when we want to compare two formulas, we have only to compare pointers instead of strings.

The Proof object returned by the prove method contains a reference to the Problem object that was passed to that method. It also has a ProofTree object, describing the tree constructed by the tableau proof, and a boolean value indicating if the tableau is closed.

A Prover object contains a reference to a Strategy object. It is the Strategy that analyses the problem and builds the proof tree. The Prover object also has a reference to a Method object. The Method object is passed by the Prover to the Strategy object that uses the Method's rules (organized in a rule structure) to try to close the tableau for the problem.

In our system we are using aspects mainly for profiling and strategy feature variation. Profiling is important for analysing the performance of each strategy and of the prover as a whole. The Profiler aspect lets us know, for instance, which rules were used and how many times each rules was applied. And, thanks to the obliviousness feature of aspect-orientation, we implemented profiling without modifying the source code that implements strategies.

In the current version of our system (a previous version is described in [Neto and Finger 2005]), we have implemented three strategies: SimpleStrategy, MemorySaverSimpleStrategy and MemorySaverStrategy (see the class hierarchy for strategies Figure 4). Each Strategy is represented as a class and the StrategyFeature aspect (see Figure 3) is used to change features of strategies that crosscut the class hierarchy for strategies. For instance, if we want to add a feature to SimpleStrategy and MemorySaverStrategy, but not to MemorySaverSimpleStrategy, we write an advice method in the StrategyFeature aspect and a pointcut to add this advice to both classes. Doing this way, we do not have to refactor the whole class hierarchy for startegies.

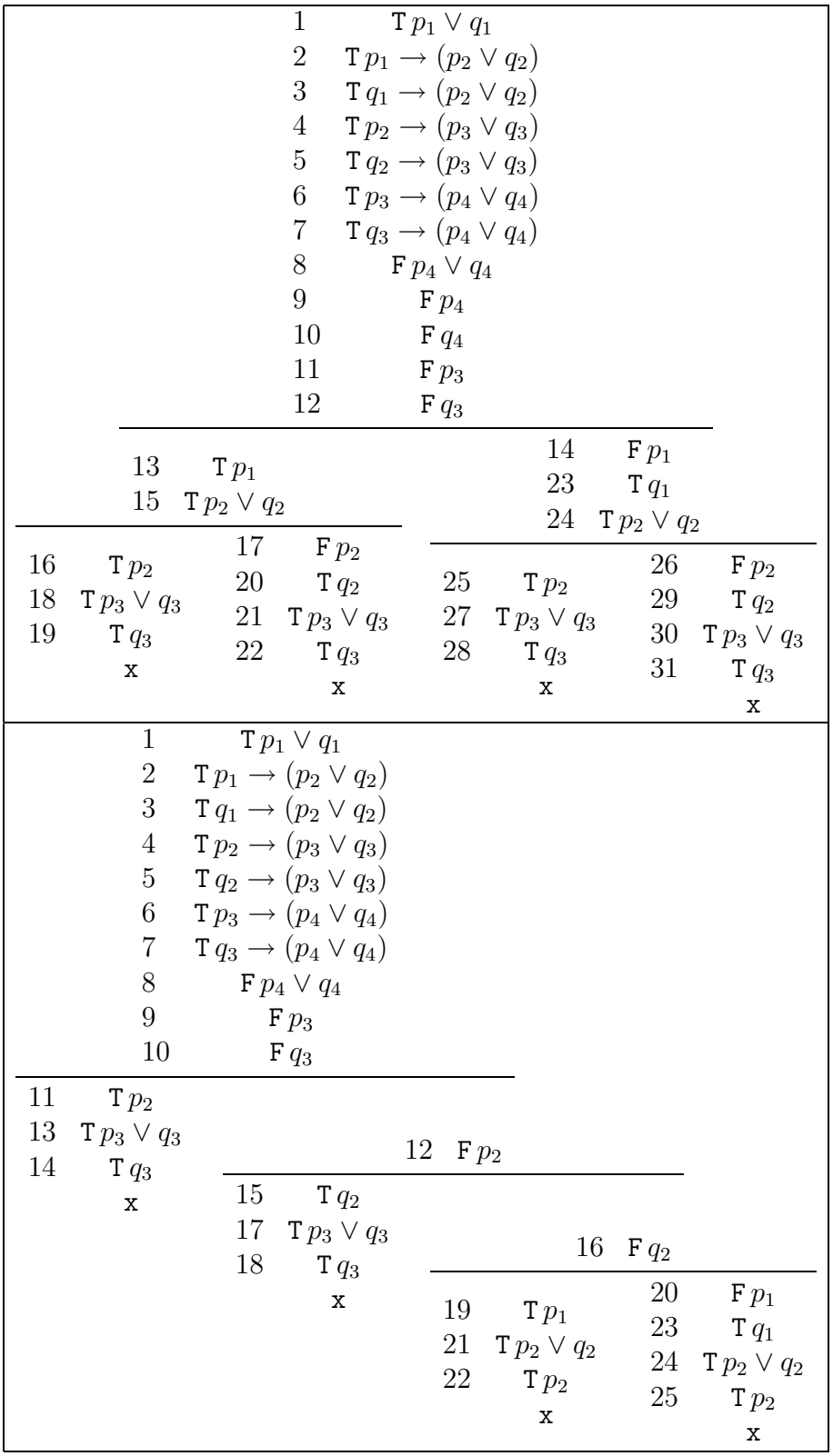

Figure 2. Two proofs of Γ_3.

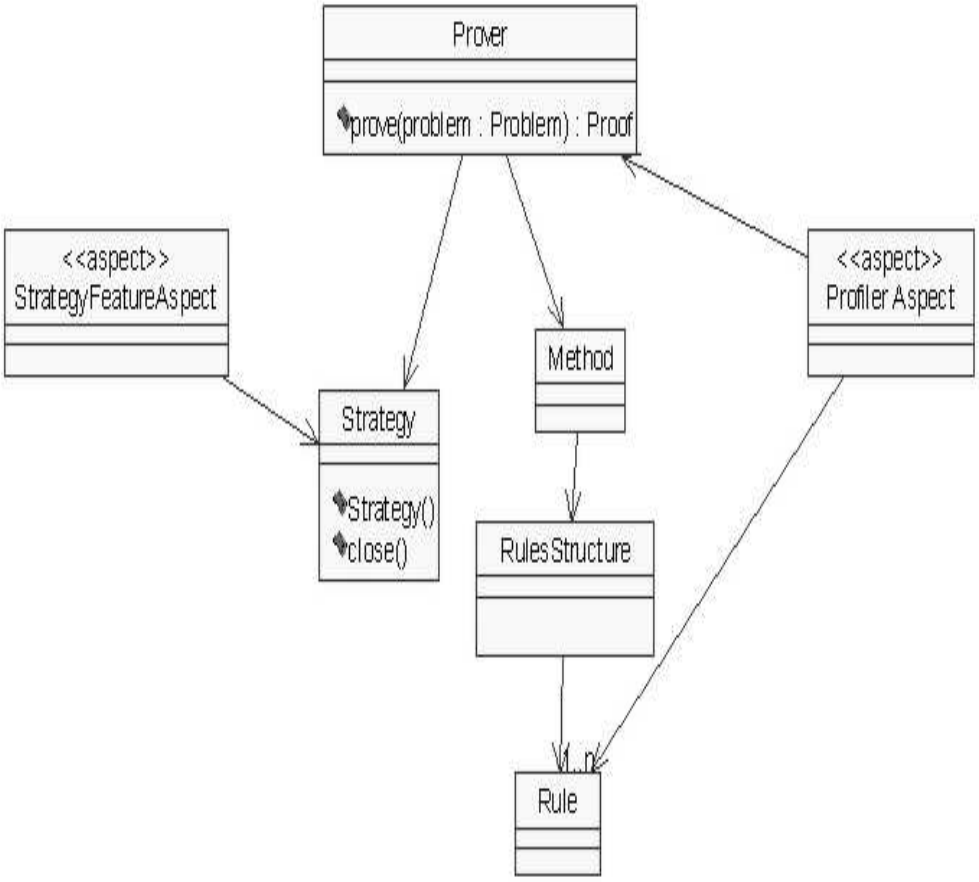

Figure 3. System class hierarchy with aspects.

4. Conclusion

In this paper we have presented some remarks about the development of a multi-strategy prover using AOP. Answering to the questions posed in the introduction, object-orientation and aspect-orientation can be used to achieve modularity in the definition of proof strategies in the following way: a proof strategy is represented as an object of a given Strategy class. Therefore, a class hierarchy such as that of Figure 4 describes the primary decomposition of strategy features. And aspects are used to change the original behaviour of a strategy. That is, the features of strategies that crosscut the primary decomposition are implemented as advice methods in an aspect.

We have implemented this prover using Java and AspectJ [Kiczales et al. 2001]. In [Neto and Finger 2005] we presented the results of the evaluation of our system for several problems using two different strategies. The next steps in our work will be to implement new strategies for clasical propositional logic using techniques adapted from DPLL procedure implementations and to extend our prover to deal with tableau systems for logics of formal inconsistency [Carnielli and Marcos 2001].

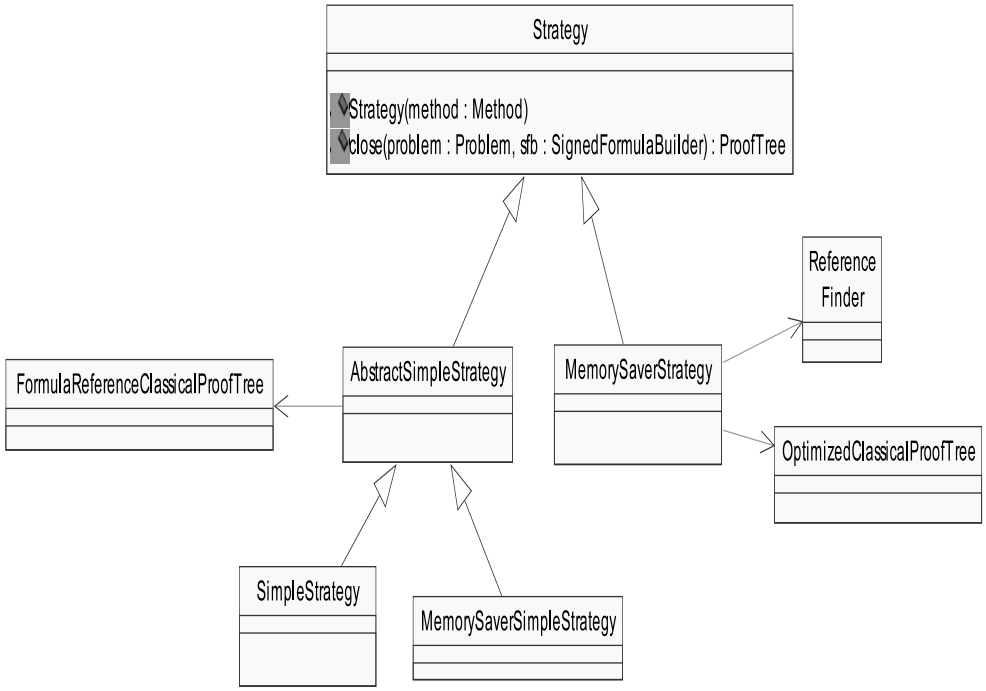

Figure 4. Class hierarchy for strategies

References

Carnielli, W. A. and Marcos, J. (2001). Tableaux for logics of formal inconsistency. In Arabnia, H. R., editor, *Proceedings of the 2001 International Conference on Artificial Intelligence (IC-AI 2001), held in Las Vegas, USA, June 2001, volume II, pages 848 852. CSREA Press, Athens GA, USA*. http://logica.rug.ac.be/∼joao/Publications/Congresses/tableauxLFIs.pdf.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms - Second Edition*. MIT Press.

D'Agostino, M. and Mondadori, M. (1994). The taming of the cut: Classical refutations with analytic cut. *Journal of Logic and Computation*, pages 285–319.

Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397.

Fariñas del Cerro, L., Fauthoux, D., Gasquet, O., Herzig, A., Longin, D., and Massacci, F. (2001). Lotrec: the generic tableau prover for modal and description logics. In *International Joint Conference on Automated Reasoning*, LNCS, page 6. Springer Verlag.

Fitting, M. (1999). Introduction. In et al., M. D., editor, *Handbook of Tableau Methods*, chapter 1, pages 1–43. Kluwer Academic Press.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Adisson-Wesley.

Gentzen, G. (1969). Investigations into logical deductions, 1935. In Szabo, M. E., editor, *The Collected Works of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355.

Neto, A. and Finger, M. (2005). Implementing a multi-strategy theorem prover. In Garcia, A. C. B. and Osório, F. S., editors, *Proceedings of the V ENIA (Encontro Nacional de Inteligência Artificial), held in São Leopoldo-RS, Brazil, July 22-29 2005*. Available at http://www.unisinos.br/_diversos/congresso/sbc2005/_dados/anais/pdf/arq0175.pdf.

Neto, A. G. S. S. (2003). An Object-Oriented Implementation of a KE Tableau Prover. Avaliable at http://www.ime.usp.br/∼adolfo.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41. Reprinted in [Siekmann and Wrightson 1983].

Siekmann, J. and Wrightson, G., editors (1983). *Automation of Reasoning: Classical Papers in Computational Logic 1957–1966*, volume 1. Springer-Verlag.

Smullyan, R. M. (1968). *First-Order Logic*. Springer-Verlag.

Soares, S. and Borba, P. (2002). AspectJ - Programação orientada a aspectos em Java. *Tutorial no SBLP 2002, 6o. Simpósio Brasileiro de Linguagens de Programação. 5 a 7 de Junho, PUC-Rio, Rio de Janeiro, Brasil*, pages 39–55.