A Revised Tactic Semantics for the Tableau Work Bench

Jack Daniel Kelly

A subthesis submitted in partial fulfillment of the degree of
Bachelor of Software Engineering at
School of Computer Science
Australian National University

October 2009

Typeset in Palatino by TEX and LATEX 2_ε.

Except where otherwise indicated, this thesis is my own original work.

Jack Daniel Kelly
30 October 2009

To my family and friends. Thanks for your support.

Acknowledgements

This project would not have been completed but for the help and support of several people. First, I would like to thank my supervisor, Dr. Rajeev Goré for suggesting such an interesting project, for his guidance throughout and for always being ready to offer advice about this project and my future career. I also thank Florian Widmann for his feedback and for the useful software tools and LaTeX macros provided along the way. I also want to thank Dr. Pietro Abate for lending his O'Caml expertise from the other side of the planet. I want to thank my friends for their support and for checking up on me when I disappeared into crunch mode. Finally, I thank my family for their support and tolerance as I saw this project through.

Abstract

The Tableau Work Bench (TWB) provides a user-friendly framework for building automated tableau-based theorem provers. Its target audience includes researchers who want to experiment with automated reasoners for calculi that they are designing. The TWB allows them to do so even with limited programming experience.

An important consideration when designing a tableau calculus for a logic is the order of rule applications. Incorrectly ordering rule applications can cause a calculus to become unsound, incomplete or non-terminating. The TWB addresses this by implementing a tactic language based on the language ”Angel” described by [Martin et al. 1996], but the semantics of this language in the context of the TWB are not completely specified and are not expressive enough in certain instances.

In this thesis, we restrict the notion of a tableau rule in order to provide well-defined semantics for rule application and tactic evaluation. We also provide a sample implementation of our rule and tactic system in O'Caml but leave its integration into the TWB as further work. We also defer the proofs that the implementation meets its specification and that the specified system is equivalent to a corresponding naïve tableau procedure.

Contents

Chapter 1

Introduction

Logical formalisms have proven to be a useful tool for many automated reasoning tasks. The satisfiability problem for classical propositional logic has been well-studied and modern SAT solvers have been effectively used for a variety of problems including dependency analysis in GNU/Linux distributions [Berre and Parrain 2008]. The model checker SMV allows users to verify properties of their systems by turning the specification into a statement in the temporal logic CTL [McMillan 2000]. CardKt [Goré and Nguyen 2000] is a proof of concept that uses multi-modal logic to securely verify the behaviour of programs running on Java Smart Cards.

Although classical propositional logic is useful for many tasks, we often want to use something more expressive without moving to first-order logic where theoremhood is undecidable. Modal and temporal logics aim for a middle ground: increased expressiveness without sacrificing decidability.

Modal logics extend the definition of formulae by adding new connectives called modalities. The modal logic K adds $\Box\varphi$, meaning "it is necessary that φ" and $\Diamond\varphi$ meaning "is is possible that φ". Temporal logics define modal operators that can reason about "future" and "past" states.

Classical (two-valued) logics are sometimes too strict to be useful: if a reasoning agent discovers a contradiction, it can no longer be trusted to say anything useful because it is able to prove anything. Belnap, for example [Belnap 1976], proposes a four-valued logic where atomic statements can take the values $\{True, False, Both, None\}$.

Tableau systems are syntactic calculi that can be used as the basis for an automated decision procedure to test theoremhood of a statement in a given logic. The decision procedure and its associated data structures can be made extremely compact: [Goré and Nguyen 2000] implemented a decision procedure on a smart card with only 512 bytes of RAM and a 32KB EEPROM for storage.

If a logician devises a new logic and an associated calculus, it is difficult for them to create an automated procedure without programming experience. Generic systems such as the Tableau Work Bench [Abate 2007b] and LoTREC [Farias del Cerro et al. 2001] have arisen to fill this need. The TWB is syntax-driven and builds a prover based on an encoding of the logician's calculi. LoTREC attempts to construct a model based on an encoding of the logic's semantics.

LoTREC has a formal semantics specified in [Gasquet et al. 2009], which means that if the rules given to LoTREC are "correct" and LoTREC implements its specification correctly, then the process will only produce "correct" results. The TWB has no formally defined semantics. Our project aims to make the first steps in rectifying this by providing a high-level description of a slightly restricted TWB.

1.1 Structure of the Thesis

Chapter 2 discusses related work and highlights the place of our work within this area.

Chapter 3 provides a brief introduction to modal logic and tableau methods, paying particular attention to the concerns which automated tableau systems need to address.

Chapter 4 introduces the Tableau Work Bench as a tool for constructing automated tableau systems. We show how the TWB addresses the concerns raised in Chapter 3 and describe some of its shortcomings.

Chapter 5 sets out our proposed semantics for a TWB-style generic tableau prover using a high-level pseudocode.

Chapter 6 uses the system described in Chapter 5 to define several provers for the modal logics K and KT, and for a bi-modal version of K. We also benchmark our provers against a standard set of benchmark data and discuss the results.

Chapter 7 presents our conclusions and sets out a scheme for further work in this area.

Chapter 2

Related Work

In this chapter, we explore related work in the area of generic automated proof systems that can handle (at least) modal propositional logics and that allow the user to easily define new logics. This is a niche area and we are aware of only two such generic systems: the Tableau Work Bench and LoTREC (Logical Tableaux Research Engineering Companion).

Systems such as the Logic Work Bench [Heuerding et al. 1996] provide optimised implementations of many common logics. Unlike the generic systems that are our focus, the LWB is difficult if not impossible for non-technical users to extend to their own logics. The LWB is written in C++ and is closed source. Without development headers, it is extremely difficult (and certainly out of reach of the average logician) to build a new logic module.

The most important feature of the LWB from our perspective is its standardised set of benchmark formulae for the logics K, KT and S4. The benchmarks are described in [Heuerding and Schwendimann 1996] and have been used in the past to benchmark both the Tableau Work Bench [Abate 2007b] and LoTREC [Gasquet et al. 2009].

2.1 The Tableau Work Bench

The Tableau Work Bench was first presented in [Abate and Goré 2003] as a meta-system for expressing tableau provers. The user defines tableau rules and specifies an overall strategy to guide the proof search.

The TWB essentially searches through a space of rooted trees, where each node is labelled with a set of logical formulae. The goal of the search is to produce a tree satisfying certain conditions, which we call a "closed tableau" (we will cover this notion in more detail later). Applying a rule to a leaf turns it into an internal node, generating new leaves based on the rule's denominators. Additional complexity is introduced because backtracking is sometimes required. If the result of a search is unsatisfactory, parts of the tree may need to be discarded and rebuilt from a different rule application or formula choice within a rule application. To manage the size of the search space, the user specifies a strategy that restricts when rules are tried, using a tactic language.

The Tableau Work Bench has since been completely rewritten, bringing several improvements:

- The input format was made friendlier for the end user. The amount of O'Caml code needed to implement simple logics is significantly reduced. Instead, "syntactic sugar" that looks like a standard presentation of tableau rules is used, improving accessibility for non-technical users.

- The custom language used to specify strategies was replaced with a clearer, more expressive one, using [Martin et al. 1996] as a base.

- The internal, hard-coded list of possible connectives was removed. The user now provides a grammar describing the formulae of their logic.

- An interface for defining sequent calculi was added.

The Angel language described in [Martin et al. 1996] is a generic language for expressing tactic programs. The only assumption Angel makes about rules is that they transform an expression from one form to another. The language consists of primitive tactics ($Skip$, $Fail$, and the rules themselves) and combinators for sequencing, alternation, recursion and a "cut" operator to restrict backtracking.

An intermediate version of the TWB was used as the basis of [Abate 2007b], which also identifies several important features of a meta-tableau system:

- A tactic language to manage non-determinism inherent in the specification of tableau calculi. While some amount of non-determinism is unavoidable, unrestricted non-determinism makes the search space too large for automated deduction.

- Histories to record information passed down through the tableau. Without histories, it is quite easy to find input that causes non-termination even for fairly simple logics.

- "Upward variables", that are the dual of histories: they collect information from the leaves of the tableau and propagate it upwards as the search procedure backtracks.

The most recent version of the TWB is described in [Abate and Goré 2009]. This system is extremely similar to the one described in [Abate 2007b]. The tactic calculus has been altered again: the "cut" operator was removed and replaced by a new type of alternation that implicitly restricts backtracking.

The basic goal of the TWB has remained unchanged throughout its various incarnations: it is designed to give users who are not strong programmers the ability to construct automated provers for their logics.

The TWB does not have a formal specification for the behaviour of the tactic system. The TWB descibed in [Abate 2007b] uses a language very similar to Angel but relies on an intuitive understanding of "success" and "failure" of tactics. The newer TWB detailed in [Abate and Goré 2009] elaborates on this slightly, but still relies on the reader's intuition to fill in blanks.

2.2 LoTREC

The other major logic-agnostic system of which we are aware is LoTREC. We explore it in some depth as it is important to show how its approach is different to that of the TWB.

2.2.1 Theory

While the TWB uses tree-based tableau and takes a purely syntactic approach to proof search, LoTREC uses the semantics of the logic being studied to define graph-based tableau. These graph tableau are described in [Castilho et al. 1997]. Instead of tableau rules that rewrite nodes in a proof tree, rules modify a rooted directed acyclic graph. Each node in the graph is associated with a set of formulae that are considered to be true at that node. The graph is started by creating a single node with the input formula and rules are applied until either a contradiction is found or a fixed point is reached.

A rule in this style of modal tableau is either a structural rule that alters the graph (by adding or removing nodes or edges), or a propagation rule that alters formulae (e.g., by moving formulae between worlds).

In many modal logics, the formula $\Diamond\varphi$ is read as "this node has a successor that makes φ true". A common structural rule for $\Diamond\varphi$ creates a new node in the graph that is a direct successor of the current node (the node that made $\Diamond\varphi$ true), and makes φ true at this new node. This point bears repeating: unlike purely syntactic methods, this calculus of graph operations attempts to build a model of the input formula directly.

Another commonly seen formula from modal logic is $\Box\varphi$, which is read "every node that is a successor of this node makes φ true". This inspires a basic propagation rule: if a node makes $\Box\varphi$ true, we pick out all of its immediate successors and require φ to be true at those nodes.

The process of applying structural and propogation rules continues until either a fixed point of all rules is reached, in which case the constructed graph is a model of the input formula or a contradiction is detected in the graph, which means that the input formula cannot be satisfied.

2.2.2 Practise

LoTREC is described in [Farias del Cerro et al. 2001] as a generic tableau prover for logics that have a "possible worlds" semantics. The authors intend it to be particularly useful for modal and description logics.

Beyond requiring that the logic has an graph-based semantics, LoTREC assumes little about the user's logic. The user is required to specify the syntax of the logical connectives, which LoTREC uses to construct a parser for input formulae.

A rule in LoTREC is a set of conditions associated with a set of actions. If every condition is satisfied, then all of the associated actions are executed. LoTREC provides a broad but fixed set of conditions and actions [Sahade 2004]. The provided conditions cover tests such as: "is there a formula matching this pattern at this node?", "is this node linked to this other node?" and the actions are what one would expect from a program that manipulates graphs ("create a new node", "link this node to this other node", "mark this expression with that flag", ...).

The specification of rules in LoTREC is restricted to the constructs provided by the authors but users can make use of "oracle programs" to handle tasks such as formula rewriting or satisfiability checking.

LoTREC has a global idea of rule application: once it decides to apply a rule, it will try to apply it to every node of a graph at once. This raises efficiency concerns in large graphs where a large fraction of the nodes could be uninteresting for any given rule.

LoTREC reduces the possibility of redundant work in two main ways. It requires the user to specify a search strategy with the `repeat`, `allRules` and `firstRule` combinators. `repeat ... end` executes the enclosed strategy until it is told to stop. `allRules ... end` tries to apply each contained strategy once. `firstRule ... end` tries to apply each contained strategy in turn, stopping once a strategy succeeds.

LoTREC also uses an event dispatching system to track which rules may be applicable and to temporarily "turn off" rules which will not achieve anything. [Gasquet et al. 2009] provides a formal semantics for this event-based mechanism and proves that it is equivalent to a naïve rewriting scheme.

2.2.3 Comparing LoTREC with the TWB

Table 2.1 provides a side-by-side comparison of the main features of both systems. It is interesting to note that the two systems are approaching the same problem from essentially opposite directions. The TWB uses syntactic methods to find a contradiction in the input, while LoTREC uses semantically-inspired rules to build a model for the input.

2.3 Our Contribution

We believe that the TWB's syntactic approach provides a more generally applicable abstraction than LoTREC's semantics-inspired approach, which is geared towards modal and description logics in particular. We also believe that the O'Caml "hooks" provided by the TWB will be more useful that LoTREC's "oracle programs" Despite this, modal logics have interesting characteristics that have motivated the features of the TWB. We therefore examine the tableau systems of several modal logics to understand the necessary features of a generic tableau system and study the TWB to find weaknesses to address. We then present a high-level pseudocode description of proof search, rule application and tactic evaluation and use this to describe provers for some modal logics.

We discuss our implementation (as an O'Caml library) of the pseudocode description and how we have constructed multiple provers for the logics K and KT. We have benchmarked these provers against the LWB's benchmark suite, which is detailed in [Heuerding and Schwendimann 1996].

Feature	LoTREC	TWB
Customisable connectives	Yes	Yes
Fundamental idea	Allow a encoding of rules drawn from a logic's semantics.	Allow a natural encoding of a tableau calculus.
Fundamental structure	Rooted directed acyclic graphs of nodes with formula sets.	Trees of nodes with formula sets and attached histories.
Search strategy	Follow the user's strategy by applying rules to the graph until one of the rules signals to stop.	Depth first search - explore nodes one at a time, applying rules to leaves according to the user's strategy, backtracking until a satisfactory result is reached.
Output	A model of the input formula, or an identification of a contradiction.	A proof of closure, or trace of rule applications that shows that no closed result could be found.
Rule application	Global - applying a rule means applying it everywhere on the graph at once.	Local - applying a rule creates new nodes to explore on that branch only.
Rule extensibility	Calls to external "oracle programs"	Hooks to embed arbitrary O'Caml code as part of a rule's condition or effect.
Histories	Basic support for "marking" a formula.	Histories are arbitrary O'Caml types (that meet certain restrictions) and can be manipulated by any user-provided function.
Strategy language	A simple strategy language for simple needs - all of the semantic information is captured by the rules.	A more expressive and complicated language to allow control over backtracking and combination of results from exploring different sub-tactics.
Formal semantics	Yes	None

Table 2.1: Feature comparison between the TWB and LoTREC

Chapter 3

Modal Logic and Tableau Methods

In this chapter, we present the syntax and semantics for two logics that motivate many of the Tableau Work Bench's features. We cover Classical Propositional Logic (CPL) and the Modal Logics K and KT. For each logic, we present the syntactic structure of its formulae and a semantics to assign meaning to the formulae. We then present syntactic derivation procedures for each logic but omit proofs of soundness and completeness as they are not new results.

This chapter is based on material from [Russell and Norvig 2003] and [D'Agostino et al. 1999].

3.1 Conventions

In this chapter, we use the following conventions:

- *Atom* stands for an infinite set of propositional atoms,
- Lowercase Roman letters $p, q, r, \ldots$ stand for individual propositional atoms,
- Lowercase Greek letters $\varphi, \psi, \ldots$ stand for individual formula, and
- Uppercase Greek letters $\Gamma, \Delta, \ldots$ stand for (finite, possibly empty) sets of formulae.

3.2 Tableau Systems

A tableau system provides a syntactic derivation procedure by deconstructing sets of formulae.

Definition 3.2.1. Tableau Rule.

A tableau rule ρ consists of a numerator $\mathcal{N}$ and either finitely many denominators $\mathcal{D}_1, \ldots \mathcal{D}_n$ separated by single vertical bars, or the symbol $\times$:

$$(\rho)\frac{\mathcal{N}}{\mathcal{D}_1 | \ldots | \mathcal{D}_n} \qquad\qquad (\rho')\frac{\mathcal{N}}{\times}$$

The numerator and each denominator are finite sets of formula shapes. Application of a rule (ρ) to a formula set Γ involves computing a unifier θ such that $\mathcal{N}\theta = \Gamma$. The denominators of the rule are then instantiated using θ to generate subgoals $\mathcal{D}_1\theta \dots \mathcal{D}_n\theta$.

(ρ') is the special case where there are no denominators: it terminates a branch if a unifier exists. We use the symbol $\times$ to clearly indicate the termination of a branch.

Definition 3.2.2. Tableau Calculus.

A Tableau Calculus (or Tableau System) is a finite set of tableau rules $\{\rho_1, \dots \rho_n\}$.

Definition 3.2.3. Tableau.

A tableau for a formula set Γ is a rooted tree of nodes where:

- Γ is the root node, and
- all children of a node are obtained from a parent by instantiating a tableau rule from the same tableau calculus.

If every leaf of a tableau is the symbol $\times$, the tableau is closed. If a tableau is not closed, it is open.

Definition 3.2.4. Invertible rule.

A rule ρ in a tableau calculus τ is invertible iff whenever there exists a closed τ-tableau for an instance of the numerator of ρ, there exist closed τ-tableaux for that same instance of each denominator.

For clarity, we will write invertible rules with a double line separating the numerator and denominator even before we have introduced the relevant inversion lemmas.

3.2.1 Automation

Tableau calculi can be used as a basis for automated deduction of logical formulae. Using the input formula φ as the root node $\Gamma = \{\varphi\}$, a depth-first search can attempt to construct the rest of a closed tableau by instantiating and exploring the denominators of rule applications. Care must be taken to ensure that the implementation of a tableau calculus does not sacrifice correctness or termination. These concerns are logic-specific and are explored in Sections 3.4.4 and 3.5.3.

3.3 Classical Propositional Logic

We first present classical propositional logic, a two-valued logic with a fairly simple tableau calculus.

3.3.1 Syntax of CPL

Definition 3.3.1. Syntax of CPL.

CPL formulae are described by the following grammar (where $p \in Atom$):

$$\begin{array}{rcl} \varphi & ::= & \top \\ & | & \bot \\ & | & p \\ & | & \neg(\varphi) \\ & | & (\varphi \wedge \varphi) \\ & | & (\varphi \vee \varphi) \\ & | & (\varphi \rightarrow \varphi) \\ & | & (\varphi \leftrightarrow \varphi) \end{array}$$

For convenience, we will often omit the parentheses and assume that the connectives $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ are listed in decreasing order of precedence.

3.3.2 Semantics of CPL

Definition 3.3.2. CPL Models and valuation of CPL formulae.

A CPL Model is a function $\vartheta : Atom \rightarrow \{True, False\}$. We recursively define the value of a CPL formula under a model as follows:

$$\begin{aligned} \vartheta(\top) &= True \\ \vartheta(\bot) &= False \\ \vartheta(\neg\varphi) &= \begin{cases} True & \text{if } \vartheta(\varphi) = False \\ False & \text{otherwise} \end{cases} \\ \vartheta(\varphi \wedge \psi) &= \begin{cases} True & \text{if } \vartheta(\varphi) = True \text{ and } \vartheta(\psi) = True \\ False & \text{otherwise} \end{cases} \\ \vartheta(\varphi \vee \psi) &= \begin{cases} True & \text{if } \vartheta(\varphi) = True \text{ or } \vartheta(\psi) = True \\ False & \text{otherwise} \end{cases} \\ \vartheta(\varphi \rightarrow \psi) &= \begin{cases} True & \text{if } \vartheta(\varphi) = False \text{ or } \vartheta(\psi) = True \\ False & \text{otherwise} \end{cases} \\ \vartheta(\varphi \leftrightarrow \psi) &= \begin{cases} True & \text{if } \vartheta(\varphi) = \vartheta(\psi) \\ False & \text{otherwise} \end{cases} \end{aligned}$$

We further extend the definition of ϑ to cover sets of formulae:

$$\vartheta(\Gamma) = \begin{cases} True & \text{if } \vartheta(\gamma) = True \text{ for every } \gamma \in \Gamma \\ False & \text{otherwise} \end{cases}$$

Thus for finite sets $\Gamma = \{\gamma_1, \gamma_2, \ldots \gamma_n\}$, $\vartheta(\Gamma) = \vartheta(\gamma_1 \wedge \gamma_2 \wedge \ldots \wedge \gamma_n)$.

Definition 3.3.3. Semantic entailment in CPL.

For a set of CPL-formulae Γ and a single CPL-formula φ, we say Γ entails φ and write $\Gamma \models \varphi$ iff for all CPL-models ϑ, if $\vartheta(\Gamma) = \mathit{True}$ then $\vartheta(\varphi) = \mathit{True}$.

Definition 3.3.4. Satisfiability in CPL.

A CPL-formula φ is satisfiable iff there exists a CPL-model ϑ such that $\vartheta(\varphi) = \mathit{True}$.

Definition 3.3.5. Validity in CPL.

A CPL-formula φ is CPL-valid iff for every CPL-model ϑ, $\vartheta(\varphi) = \mathit{True}$.

Thus a CPL-formula φ is valid iff $\emptyset \models \varphi$. Further, a CPL-formula φ is valid iff its negation $\neg\varphi$ is not CPL-satisfiable.

3.3.3 A Tableau System for CPL

Our procedure requires the input formulae to be in negation normal form, which we define below.

Definition 3.3.6. Negation Normal Form of a CPL formula.

A CPL-formula φ is in negation normal form if it contains no $\rightarrow$ or $\leftrightarrow$ connectives and every occurrence of $\neg$ is adjacent to an atom. The negation normal form of a CPL formula is computed as follows:

$$\begin{array}{rcl}
nnf(\top) & = & \top \\
nnf(\neg\top) & = & \bot \\
nnf(\bot) & = & \bot \\
nnf(\neg\bot) & = & \top \\
nnf(p) & = & p \\
nnf(\neg p) & = & \neg p \\
nnf(\neg\neg\varphi) & = & nnf(\varphi) \\
nnf(\varphi \wedge \psi) & = & nnf(\varphi) \wedge nnf(\psi) \\
nnf(\neg(\varphi \wedge \psi)) & = & nnf(\neg\varphi) \vee nnf(\neg\psi) \\
nnf(\varphi \vee \psi) & = & nnf(\varphi) \vee nnf(\psi) \\
nnf(\neg(\varphi \vee \psi)) & = & nnf(\neg\varphi) \wedge nnf(\neg\psi) \\
nnf(\varphi \rightarrow \psi) & = & nnf(\neg\varphi \vee \psi) \\
nnf(\neg(\varphi \rightarrow \psi)) & = & nnf(\varphi) \wedge nnf(\neg\psi) \\
nnf(\varphi \leftrightarrow \psi) & = & nnf((\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)) \\
nnf(\neg(\varphi \leftrightarrow \psi)) & = & nnf(\neg(\varphi \rightarrow \psi)) \vee nnf(\psi \rightarrow \varphi)
\end{array}$$

The following lemma shows that restriction to negation normal form does not limit the expressive power of our tableau calculus:

Lemma 3.3.7. Every CPL-formula φ has an equivalent $nnf(\varphi)$ in negation normal form such that $\emptyset \vDash \varphi \leftrightarrow nnf(\varphi)$.

Definition 3.3.8. The Tableau Calculus $\mathbb{CPL}$

Given the following tableau rules:

$$(\bot)\frac{\bot; Z}{\times} \qquad (Id)\frac{p; \neg p; Z}{\times} \qquad (\wedge)\frac{\varphi \wedge \psi; Z}{\varphi; \psi; Z} \qquad (\vee)\frac{\varphi \vee \psi; Z}{\varphi; Z | \psi; Z}$$

We define the tableau calculi $\mathbb{CPL} = \{(\bot), (Id), (\wedge), (\vee)\}$.

The root node of these tableau is a set of $\mathbb{CPL}$-formulae where each formula is in negation normal form.

The rules of a tableau calculus should preserve satisfiability downwards: if the numerator of a rule is satisfiable, then at least one of the denominators is satisfiable. The $(\vee)$ rule says that for a disjunction $\varphi \vee \psi$ to be satisfiable, then at least one of the disjuncts must be satisfiable. Applying a rule with the denominator $\times$, such as the (Id) rule, shows that the numerator is unsatisfiable. A closed tableau should therefore be equivalent to showing that the root is not satisfiable. This idea is formalised by the notions of soundness and completeness.

Lemma 3.3.9. All of the rules in $\mathbb{CPL}$ are invertible.

We omit the proof as it is not a new result.

Definition 3.3.10. Soundness of CPL-tableau.

A tableau calculus for CPL is sound iff:
If there is a closed tableau for $\{\varphi\}$ then φ is not satisfiable.

Definition 3.3.11. Completeness of CPL tableau.

A tableau calculus for CPL is complete iff:
If φ is not satisfiable then there is a closed tableau for $\{\varphi\}$.

$\mathbb{CPL}$ is sound and complete, but we omit the proofs for the sake of brevity.

Example 3.3.12. Example closed $\mathbb{CPL}$-tableau.

We show that that formula $p \wedge (q \wedge (\neg p \vee \neg q))$ is unsatisfiable by constructing a closed $\mathbb{CPL}$-tableau:

$$\begin{array}{c} p \wedge (q \wedge (\neg p \vee \neg q)) \\ \hline p; q \wedge (\neg p \vee \neg q) \quad (\wedge) \\ \hline p; q; \neg p \vee \neg q \quad (\wedge) \\ \hline \begin{array}{cc} \dfrac{p; q; \neg p}{\times}\,(Id) & \dfrac{p; q; \neg q}{\times}\,(Id) \end{array} \quad (\vee) \end{array}$$

Example 3.3.13. The $(\vee)$ rule is invertible: whenever $\{\varphi \vee \psi\} \cup Z$ has a closed tableau, $\{\varphi\} \cup Z$ and $\{\psi\} \cup Z$ have closed tableau. Intuitively, we are not losing any information moving from the numerator to the denominator.

An important feature of invertible rules is the order in which they are applied does not matter. Given a formula set $\{\neg q \wedge \neg p, p \vee q\}$, it does not matter if we apply the $(\wedge)$ rule or the $(\vee)$ rule first: there are closed tableaux for either case. When applying an invertible rule to a formula set, we are therefore able to commit to a unifier without worrying about losing completeness.

3.3.4 Computing Validity and Entailment

Tableau procedures give us a method to test if a formula is unsatisfiable. We can use this to test if a formula is valid by testing if its negation is unsatisfiable. We can syntactically test for semantic entailment $\{\gamma_1, \gamma_2, \ldots \gamma_n\} \models \varphi$ by testing if $\gamma_1 \wedge \gamma_2 \wedge \ldots \wedge \gamma_n \wedge \neg\varphi$ has a closed tableau.

3.3.5 Considerations for Automation

There are three types of decisions that the search procedure needs to make as it applies rules. First, it needs to decide which rule to apply to a formula set. Second, it needs to decide on a unifier to use to instantiate the denominators of a node. Third, it needs to decide which denominator to explore.

Automating $\mathbb{CPL}$ is fairly straightforward as all the rules are invertible. We cannot get to a "dead end" state where no rules are applicable and yet we have missed out on finding a closed tableau. However, an automated procedure needs to avoid redundant work as well as be correct. It therefore makes the most sense to try to apply the $(\bot)$ and (Id) rules first, then the $(\wedge)$ rule and branch using the $(\vee)$ rule as a last resort. The order of rule applications becomes important for correctness once non-invertible rules are introduced, such as the (K) rule in Section 3.4.3.

The other concern with automated systems is ensuring that the procedure will terminate for any input formula set Γ. In $\mathbb{CPL}$, each application of the $(\wedge)$ or $(\vee)$ rules removes one connective from Γ, whilc $(\bot)$ and (Id) instantly terminate. Because we require Γ to be finite, there cannot be an infinite number of connectives. Eventually there will come a point where it is impossible to apply the $(\wedge)$ or $(\vee)$ rules. The resulting tableau is then open or can be closed by applying the $(\bot)$ or (Id) rules.

3.4 Propositional Modal Logic

We now present a logic called K, which can reason about "worlds" that are connected in some way. The value of atoms can change from world to world and we introduce connectives for reasoning about formulae in other worlds.

3.4.1 Syntax of K

Definition 3.4.1. Syntax of K.

K formulae are described by the following grammar (where $p \in Atom$):

$$\begin{array}{rcl} \varphi & ::= & \top \\ & | & \bot \\ & | & p \\ & | & \neg(\varphi) \\ & | & \Box(\varphi) \\ & | & \Diamond(\varphi) \\ & | & (\varphi \wedge \varphi) \\ & | & (\varphi \vee \varphi) \\ & | & (\varphi \rightarrow \varphi) \\ & | & (\varphi \leftrightarrow \varphi) \end{array}$$

As before, we will often omit the parentheses. $\neg, \Box$ and $\Diamond$ have equal highest precedence, followed by $\wedge, \vee, \rightarrow$ and $\leftrightarrow$.

3.4.2 Semantics of K

Definition 3.4.2. Kripke Frame.

A Kripke Frame is a directed graph $\langle W, R\rangle$, where W is a non-empty set of worlds and $R \subseteq W \times W$ is a binary relation over W. We write $w_1 R w_2$ to mean $(w_1, w_2) \in R$.

Definition 3.4.3. Valuation on a Kripke Frame.

A valuation on a Kripke frame is a function $\vartheta : W \times Atom \rightarrow \{True, False\}$ that describes the truth value of every atom at every world.

Definition 3.4.4. Kripke Model.

A Kripke Model is a triple $\langle W, R, \vartheta\rangle$ where $\langle W, R\rangle$ is a Kripke frame and ϑ is a valuation on W.

Definition 3.4.5. Valuation of formulae.

We recursively extend the definition of ϑ to valuations of formulae:

$$
\begin{aligned}
\vartheta(w, \top) &= \mathit{True} \\
\vartheta(w, \bot) &= \mathit{False} \\
\vartheta(w, \neg\varphi) &= \begin{cases} \mathit{True} & \text{if } \vartheta(w,\varphi) = \mathit{False} \\ \mathit{False} & \text{otherwise} \end{cases} \\
\vartheta(w, \varphi \wedge \psi) &= \begin{cases} \mathit{True} & \text{if } \vartheta(w,\varphi) = \mathit{True} \text{ and } \vartheta(w,\psi) = \mathit{True} \\ \mathit{False} & \text{otherwise} \end{cases} \\
\vartheta(w, \varphi \vee \psi) &= \begin{cases} \mathit{True} & \text{if } \vartheta(w,\varphi) = \mathit{True} \text{ or } \vartheta(w,\psi) = \mathit{True} \\ \mathit{False} & \text{otherwise} \end{cases} \\
\vartheta(w, \varphi \to \psi) &= \begin{cases} \mathit{True} & \text{if } \vartheta(w,\varphi) = \mathit{False} \text{ or } \vartheta(w,\psi) = \mathit{True} \\ \mathit{False} & \text{otherwise} \end{cases} \\
\vartheta(w, \varphi \leftrightarrow \psi) &= \begin{cases} \mathit{True} & \text{if } \vartheta(w,\varphi) = \vartheta(w,\psi) \\ \mathit{False} & \text{otherwise} \end{cases} \\
\vartheta(w, \Box\varphi) &= \begin{cases} \mathit{True} & \text{if } \vartheta(v,\varphi) = \mathit{True} \text{ for every } v \in W \text{ with } wRv \\ \mathit{False} & \text{otherwise} \end{cases} \\
\vartheta(w, \Diamond\varphi) &= \begin{cases} \mathit{True} & \text{if } \vartheta(v,\varphi) = \mathit{True} \text{ for some } v \in W \text{ and } wRv \\ \mathit{False} & \text{otherwise} \end{cases}
\end{aligned}
$$

Definition 3.4.6. Semantic forcing on worlds and models.

Let $\mathcal{M} = \langle W, R, \vartheta \rangle$. For a world $w \in W$ and a K-formula φ, we say w forces φ and write $w \Vdash \varphi$ iff $\vartheta(w, \varphi) = \mathit{True}$. We say that the model $\mathcal{M}$ forces φ and write $\mathcal{M} \Vdash \varphi$ iff for every $w \in W$, $w \Vdash \varphi$.

If Γ is a set of K-formula, we write $\mathcal{M} \Vdash \Gamma$ iff for every $\gamma \in \Gamma$, $\mathcal{M} \Vdash \gamma$.

Definition 3.4.7. Semantic entailment in K.

Let $\mathcal{K}$ be the class of all Kripke frames, Γ be a set of K-formulae and φ a K-formula. We say Γ entails φ and write $\Gamma \vDash \varphi$ iff for every model $\mathcal{M} \in \mathcal{K}$, $\mathcal{M} \Vdash \Gamma$ implies $\mathcal{M} \Vdash \varphi$.

Definition 3.4.8. Satisfiability in K.

A K-formula φ is K-satisfiable iff there exists a Kripke model $\mathcal{M} = \langle W, R, \vartheta \rangle$ and a world $w \in W$ such that $w \Vdash \varphi$.

Definition 3.4.9. Validity in K.

A K-formula φ is K-valid iff for every Kripke model $M = \langle W, R, \vartheta \rangle$, $M \Vdash \varphi$.

Thus a K-formula φ is K-valid iff $\emptyset \vDash \varphi$. Further, a K-formula φ is K-valid iff its negation $\neg\varphi$ is not K-satisfiable.

3.4.3 A Tableau System for K

As with the tableau system for CPL, our tableau system for K require the input formulae to be in negation normal form.

Definition 3.4.10. Negation Normal Form of a K formula.

A K-formula φ is in negation normal form if it contains no $\rightarrow$ or $\leftrightarrow$ connectives and every occurrence of $\neg$ is adjacent to an atom. The negation normal form of a K formula is computed as follows:

$$\begin{array}{rcl}
nnf(\top) & = & \top \\
nnf(\neg\top) & = & \bot \\
nnf(\bot) & = & \bot \\
nnf(\neg\bot) & = & \top \\
nnf(p) & = & p \\
nnf(\neg p) & = & \neg p \\
nnf(\neg\neg\varphi) & = & nnf(\varphi) \\
nnf(\varphi \wedge \psi) & = & nnf(\varphi) \wedge nnf(\psi) \\
nnf(\neg(\varphi \wedge \psi)) & = & nnf(\neg\varphi) \vee nnf(\neg\psi) \\
nnf(\varphi \vee \psi) & = & nnf(\varphi) \vee nnf(\psi) \\
nnf(\neg(\varphi \vee \psi)) & = & nnf(\neg\varphi) \wedge nnf(\neg\psi) \\
nnf(\varphi \rightarrow \psi) & = & nnf(\neg\varphi \vee \psi) \\
nnf(\neg(\varphi \rightarrow \psi)) & = & nnf(\varphi) \wedge nnf(\neg\psi) \\
nnf(\varphi \leftrightarrow \psi) & = & nnf((\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)) \\
nnf(\neg(\varphi \leftrightarrow \psi)) & = & nnf(\neg(\varphi \rightarrow \psi)) \vee nnf(\psi \rightarrow \varphi) \\
nnf(\Box\varphi) & = & \Box(nnf(\varphi)) \\
nnf(\neg\Box\varphi) & = & \Diamond(nnf(\neg\varphi)) \\
nnf(\Diamond\varphi) & = & \Diamond(nnf(\varphi)) \\
nnf(\neg\Diamond\varphi) & = & \Box(nnf(\neg\varphi))
\end{array}$$

As with CPL, the following lemma shows that restricting the tableau calculus to formulae in negation normal form does not sacrifice expressive power:

Lemma 3.4.11. Every K-formula φ has an equivalent formula $nnf(\varphi)$ in negation normal form such that $\emptyset \vDash \varphi \leftrightarrow nnf(\varphi)$.

Definition 3.4.12. The Tableau Calculus $\mathbb{K}$.

Given the following tableau rules:

$$(\bot)\frac{\bot; Z}{\times} \qquad (Id)\frac{p; \neg p; Z}{\times} \qquad (\wedge)\frac{\varphi \wedge \psi; Z}{\varphi; \psi; Z} \qquad (\vee)\frac{\varphi \vee \psi; Z}{\varphi; Z \mid \psi; Z}$$

$$(K)\frac{\Diamond\varphi; \Box X; Z}{\varphi; X}$$

We define the tableau calculus $\mathbb{K} = \mathbb{CPL} \cup \{(K)\}$. As with our tableau calculi for CPL, we require the root node to be in negation normal form.

Definition 3.4.13. Soundness of K-tableau.

A tableau calculus for K is sound iff:
If there is a closed tableau for $\{\varphi\}$ then φ is not satisfiable.

Definition 3.4.14. Completeness of K-tableau.

A tableau calculus for K is complete iff :
If φ is not satisfiable there exists a closed tableau for $\{\varphi\}$.

$\mathbb{K}$ is also sound and complete, but we again omit the proofs for brevity.

Example 3.4.15. The (K) rule is not invertible: the formulae in the set Z are lost when moving to the denominator. The set $\{\Diamond p, \bot\}$ has a closed tableau:

$$\frac{\Diamond p; \bot}{\times}\ (\bot)$$

But applying the (K) rule instead of the $(\bot)$ rule results in a new set which does not have a closed tableau:

$$\frac{\Diamond p; \bot}{p}\ (K)$$

No rule is applicable to the denominator $\{p\}$. This tableau is open and yet our initial formula set was unsatisfiable.

3.4.4 Considerations for Automation

There are two issues to consider when automating the derivation procedure for K. Both issues are cause by the non-invertibility of the (K) rule.

First, whenever a search procedure applies the K rule, it cannot know in advance which $\Diamond$-formula, if any, will close. Consider the set $\{\Diamond p, \Diamond\bot\}$. If (K) is applied to $\Diamond p$, the result is an open tableau, but if (K) is applied to $\Diamond\bot$, the tableau will close after applying $(\bot)$. A backtracking mechanism is required at each application of the (K) rule to test every $\Diamond$-formula and to close if any of the choices generate a closed tableau. More generally, an invertible rule such as $(\wedge)$ or $(\vee)$ allows us to commit to the formula choice, while a non-invertible rule like (K) requires us to backtrack across formula choices to find a closed tableau.

As we have seen in Example 3.4.15, the (K) rule loses information by discarding the set Z from the numerator. We need to apply the $(\wedge)$ and $(\vee)$ rules as much as possible before applying (K) for two reasons:

- To detect any closed tableau that do not involve $\Box$- or $\Diamond$-formulae, before these formulae are discarded by the (K) rule, and

- To ensure that all $\Box$-formulae in the formula set are "detected" by the (K) rule.

Therefore, a decision procedure must first "saturate" the formula set by applying the $(\wedge)$ and $(\vee)$ rules as many times as possible and only then apply the (K) rule. The result is that by the time we apply the (K) rule, the set Z can only contain atoms, negated atoms, $\top$ or $\bot$.

Termination of the $\mathbb{K}$ procedure is ensured by a similar argument to the argument used for $\mathbb{CPL}$. First, we need to define the modal depth of a formula:

Definition 3.4.16. Modal depth of a formula.

The modal depth of a formula is defined by the following equalities:

$$\begin{aligned}
modaldepth(\bot) &= 0\\
modaldepth(\top) &= 0\\
modaldepth(p) &= 0\\
modaldepth(\neg\varphi) &= modaldepth(\varphi)\\
modaldepth(\Box\varphi) &= 1 + modaldepth(\varphi)\\
modaldepth(\Diamond\varphi) &= 1 + modaldepth(\varphi)\\
modaldepth(\varphi \wedge \psi) &= \max\{modaldepth(\varphi), modaldepth(\psi)\}\\
modaldepth(\varphi \vee \psi) &= \max\{modaldepth(\varphi), modaldepth(\psi)\}\\
modaldepth(\varphi \rightarrow \psi) &= \max\{modaldepth(\varphi), modaldepth(\psi)\}\\
modaldepth(\varphi \leftrightarrow \psi) &= \max\{modaldepth(\varphi), modaldepth(\psi)\}
\end{aligned}$$

Because we operate on finite sets Γ and formulae cannot have infinite modal depth, we know that $\arg\max_{\gamma\in\Gamma} modaldepth(\gamma)$ is finite. Each "saturation step" is essentially tableau search using $\mathbb{CPL}$, which we know will terminate. When the (K) rule is applied, each formulae is either removed or has its modal depth reduced by one. We therefore cannot apply the (K) rule an infinite number of times. Becase we cannot generate infinitely many rule applications, our procedure must terminate.

3.5 Beyond K

We now present two straightforward extensions to K which have interesting implications for their automated deduction procedures.

3.5.1 Bi-modal K

A bimodal version of K adds an additional relation to the Kripke frames $\langle W, R, R'\rangle$. We add new connectives $\blacklozenge$ and $\blacksquare$ to our definition of well-formed formulae. The formulae $\blacklozenge\varphi$ and $\blacksquare\varphi$ have identical semantics to $\Diamond\varphi$ and $\Box\varphi$ except that they operate on R' instead of R. Our calculus $\mathbb{K}$ can be extended to bimodal K by creating analogous versions of the (K) rule to deconstruct $\blacksquare$ and $\blacklozenge$ formulae.

3.5.2 KT

We can also extend K by placing additional constraints upon the reachability relation used in the Kripke frames. Requiring the relation to be reflexive yields the logic KT. In KT, every node is its own successor and so the axiom schema $\Box\varphi \rightarrow \varphi$ is valid for every formula φ. This inspires the rule (T):

$$(T)\frac{\Box\varphi; Z}{\Box\varphi; \varphi; Z}$$

Lemma 3.5.1. The rule (T) is invertible.

As with the other invertibility results, the proof is omitted as it is not novel.

Adding this rule to $\mathbb{K}$ yields the calculus $\mathbb{KT} = \mathbb{K} \cup \{(T)\}$. $\mathbb{KT}$ is sound and complete but we omit the proof as it is not a new result.

3.5.3 Considerations for Automation

Automating bimodal K is fairly straightforward. The issues are mostly the same as for automating $\mathbb{K}$ except that we now have two noninvertable rules: $(K\Diamond)$ and $(K\blacklozenge)$. Automating KT presents some additional complications that are described shortly.

3.5.3.1 Bi-modal K

As with our automated system for $\mathbb{K}$, we first need to saturate the formula set by applying $(\wedge)$ and $(\vee)$ as much as possible. After that, we need to choose to apply either $(K\Diamond)$ or $(K\blacklozenge)$. Consider the formula set $\{\Diamond p, \Diamond q, \blacklozenge r, \blacklozenge\bot\}$. Because our procedure cannot know in advance which $\Diamond$- or $\blacklozenge$-formula will close, we must be able to apply the $(K\Diamond)$ rule on both $\Diamond p$ and $\Diamond q$ and also be able to apply the $(K\blacklozenge)$ rule on $\blacklozenge r$ and $\blacklozenge\bot$. If we apply $(K\blacklozenge)$, we can generate a closed tableau, but if we apply $(K\Diamond)$ we cannot. We must therefore be able to backtrack at the rule-choice level of our search procedure when faced with a choice between non-invertible rules. If one of these rules fails to generate a closed tableau after trying all possible formula choices, we must backtrack to this choice point and continue the search by applying another rule and trying all its formula choices.

Termination for bi-modal K can be shown by a similar argument to the reasoning for standard K.

3.5.3.2 KT

For KT, the situation is a little more complicated. Consider the formula set $\{\Box p\}$. We can apply the (T) rule to get the denominator $\{p, \Box p\}$. A naïve algorithm could run forever by repeatedly applying the (T) rule to no effect:

$$\cfrac{\cfrac{\cfrac{\Box p}{p;\Box p}\,(T)}{p;\Box p}\,(T)}{\vdots}\,(T)$$

This is the simplest possible case; the infinite chain of applications cannot be detected with a lookbehind of fixed size. A terminating implementation must maintain a history of forumulae H that have had the (T) rule applied to them, and not apply (T) to the same formula twice. Our revised (T) rule is as follows:

$$(T)\frac{\Box\varphi; Z}{\varphi; \Box\varphi; Z \quad - \quad H : H \cup \{\Box\varphi\}}\ \Box\varphi \notin H$$

To the right of the rule we have its *side condition*: a condition which must be satisfied before the rule can be applied. In the denominator, we specify an *action* to the right of the $-$. If we do not specify an action, we assume that the history is copied unchanged. The action is local: it only updates the history on its subtree. The action and side condition together prevent the above infinite loop by preventing the application of (T) to the same formula multiple times. This history must be cleared after each application of the (K) rule, because the old $\Box$-formulae have now been "reduced" and new $\Box$-formulae (from inside $\Diamond\varphi$) may exist and need "unpacking". Our updated (K) rule is presented below:

$$(K)\frac{\Diamond\varphi; \Box X; Z}{\varphi; X \quad - \quad H : \emptyset}$$

Once we prevent infinite applications of (T), we can show termination with a straightforward argument using the maximum modal depth of the input formula set. We will explore side conditions and actions in detail in later chapters.

Chapter 4

The Tableau Work Bench

4.1 Background

The tableau work bench is a generic framework for building tableau-based theorem provers. Developed in 2007 as part of a PhD project, the TWB is designed to help "lazy logicians" produce naïve provers for their favourite logics without being "real programmers".

The TWB consists of two major parts: a front-end compiler that desugars the input file into O'Caml code and a back-end that implements the generic depth-first tableau search procedure and supporting data structures.

4.2 Non-Determinism

[Abate and Goré 2009] identifies and addresses three types of non-determinism in the search procedure:

Node-choice: determining which formula set to expand next in the search,

Rule-choice: deciding which rule to apply to a formula set, and

Formula-choice: deciding how to instantiate the schemas in a rule's numerator.

Node choice is handled by a depth-first search: when a rule is applied, the leftmost denominator instance is searched first. Rule choice is left to the user through the specification of a strategy. Formula choice is only important if the rule being applied is not invertible. For non-invertible rules, the TWB backtracks over rule instances until one instance closes or all instances have been generated and found to be open.

4.3 Source Format

The input to the TWB compiler is an O'Caml source file which is parsed by an extended version of the O'Caml parser [1]. This extended syntax is designed to resemble the traditional presentation of tableau rules. The desugared input file (now O'Caml

[1]O'Caml comes with a tool for extending the syntax of the language called camlp4.

code) is then compiled by the O'Caml compiler and linked against the back-end library. Figure 4.1 defines a prover for the modal logic K. The user defines a grammar describing the formulae of the logic (using the `GRAMMAR` keyword), a number of tableau rules to operate on sets of such formulae (using the `RULE` keyword) and a strategy to guide the rule application, using the `STRATEGY` keyword.

The definition of the "K" rule parallels the definition in Section 3.4.3 but the definitions of the "Id", "False", "And" and "Or" rules have been compressed to fit on a single line. An invertible rule is denoted by separating a rule's numerator and denominator with a line made of at least two "=" symbols. A non-invertible rule uses a line of at least two "-" symbols.

As we have seen before, invertibility corresponds to commiting to a formula choice. The user is assumed to have proved an invertibility lemma for any rules that have been declared invertible.

The TWB allows a second type of alternation in rule denominators. The single $|$ symbol is used to denote that the numerator is closed if every denominator is closed. The $||$ is the dual of $|$: the numerator closes if some denominator closes. This is used to implement the recursive version of the (K) rule given in Figure 4.2, which moves the alternation from the formula choice into the rule's denominator. Notice that we use the === line instead of the `---` line from the original version of K. Because the `KREC` rule does not discard the other <>-formulae, we can declare it to be "invertible" as our strategy ensures that every <>-formula will be examined by an application of `KREC`. For example, applying `KREC` to the set $\{\Diamond p; \Diamond\bot\}$ will find a closed tableau by first trying $\Diamond p$ (say) and then recursing on to the set $\{\Diamond\bot\}$ when it fails to close. The next application of `KREC` will generate the formula $\bot$, causing a closed result.

To implement logics such as KT from Section 3.5, we need histories and the ability to attach conditions to rules. Figure 4.3 demonstrates how the TWB presents side-conditions and actions to the user. We define a module `FormulaSet` that uses the TWB's builtin generic set functor[2]. Within the `HISTORIES` declaration we declare one history called `BOXES` that is a new, empty `FormulaSet.set`. The `COND` directive on the `T` rule means that it will only apply if the formula that matches `A` is not already in `BOXES`. When the rule applies, the `ACTION` directive causes whatever matched `A` to be added to `BOXES` on that branch.

The `STRATEGY` directive describes the strategy to guide rule applications. The strategy syntax is derived from the tactic calculus presented in [Martin et al. 1996]. In [Abate and Goré 2009], the following kinds of tactics are described:

- Tableau rules.
- *Skip* and *Fail*. *Skip* does nothing, successfully, behaving like a rule that is always applicable and copies the numerator unchanged into the denominator. *Fail* behaves like a rule that is never applicable.
- Deterministic alternation, written $t_1!t_2$. Deterministic alternation commits to

[2]A functor is O'Caml's method of enabling generic programming. They serve a similar role to the templated container classes in the C++ STL, or Java's generic collections.

t_1 if the first rule of t_1 is applicable. If t_1 is selected, the procedure will not backtrack to try t_2.

- Alternation, written $t_1||t_2$. If the tactic t_1 is successful, then the alternation behaves like t_1. If not, then the search backtracks and applies t_2.
- Sequencing, written $t_1;t_2$. Applying $t_1;t_2$ to a node N is meant to be equivalent to $t_2(t_1(N))$.
- Repetition, written t^*. t^* is defined to be $\mu X.(t;X|Skip)$. μ is a least fixed point operator: the X inside the expression is equivalent to the entire expression.

4.4 Shortcomings

The version of the TWB described in [Abate and Goré 2009] has multiple shortcomings as a specification, mainly regarding rule application and the evaluation of tactics:

- A closed tableau is identified with success and an open tableau with failure but a rule is specified to "succeed" if it is applicable. For rules that do not close their branch of the tableau, this is obviously incorrect.
- Rule application is never properly defined. Presumably when a sequence $t_1;t_2$ is applied to a node, t_1 will generate some subtree of the tableau and t_2 should be applied to each of the leaves. It is not clear how backtracking should behave if t_2 fails. Ideally the search should backtrack into any alternation in t_1 and try a different alternative, but this is not specified at all.
- Tactic alternation is defined as a $||$ operator, but the expansion of $*$ and the sample code both use $|$ for tactic alternation, which is never defined.
- Operator meanings are inconsistent. $|$ in the context of a denominator requires every denominator to be closed for the numerator to be closed. $|$ in the context of a tactic requires one of the tactics to be closed for the application of the tactic to close.
- The alternations offered by different areas of the search are inconsistent. Within a denominator, the procedure can try alternatives until one alternative is open or until one is closed. Within a tactic, the procedure can try alternatives until one is closed or commit to the first choice. When choosing a formula partition, the options are the same as for a tactic.

```
CONNECTIVES [ "~";"&";"v";"->";"<->";"<>";"[]" ]
GRAMMAR
formula :=
     ATOM | Verum | Falsum
    | formula & formula
    | formula v formula
    | formula -> formula
    | formula <-> formula
    | [] formula
    | <> formula
    | ~ formula
;;
expr := formula ;;
END

open Twblib
open Klib

TABLEAU
  RULE K
  { <> A } ; [] X ;  Z
  --------------------
          A ; X
  END

  RULE Id { a } ; { ~ a } === Close END
  RULE False Falsum === Close END
  RULE And { A & B } === A ; B END
  RULE Or { A v B } === A | B END
END

STRATEGY :=
    let sat = tactic ( (Id ! False ! And ! Or) ) in
    tactic ( ((sat)* ; K )* )
PP := List.map nnf
NEG := List.map neg
MAIN
```

Figure 4.1: TWB definition of a prover for the modal logic K [Abate 2007a].

```
RULE KREC
  { <> A } ; [] X ; <> Y ; Z
  ==========================
      A ; X || <> Y ; [] X
END
```

Figure 4.2: A recursive definition of the (K) rule for the logic K [Abate 2007a].

```
...

module FormulaSet = TwbSet.Make(
    struct
        type t = formula
        let to_string = formula_printer
        let copy s = s
    end
)

HISTORIES
BOXES    : FormulaSet.set := new FormulaSet.set
END

...

RULE T
  { [] A }
  ========
  A ; [] A
  COND notin(A, BOXES)
  ACTION [ BOXES := add(A,BOXES) ]
END

...
```

Figure 4.3: TWB definition of the (T) rule for the logic KT [Abate 2007a].

Chapter 5

Proposed Semantics for the TWB

In this chapter, we present a new generic tableau search procedure. We aim to present our procedure in a way that suggests a straightforward implementation. We make no specific assumptions about the logic, formula structures or history structures used. Modern programming languages often have a "generic programming" facility that can be used to achieve this, such as C++'s templates [Stroustrup 1991], Java's generics [Gosling et al. 2005] or O'Caml's functors [Leroy et al. 2008].

At certain points in this chapter, we find an algorithmic specification more natural than a purely declarative approach. We use a pseudocode that should be familiar to most readers who have a programming background.

Our procedure is inspired by and broadly similar to the ill-specified search from Chapter 4: We assume a set of rules and a tactic to guide their application.

5.1 Assumptions and Conventions

We assume the following features are specified by the user's logic:

- *Formulae* is a set of all well-formed formulae of the user's logic.
- *Schemas* is a set containing all valid schemas of the user's logic.
- *Var* is a set of variables that appear in schemas.
- *Histories* is a set of all possible history values.
- $\mathit{partitions}(\mathit{schema}, \Gamma)$ is a procedure that computes all partitions of a formula set Γ that match *schema*. A partition is a function $\pi : \mathit{Var} \to \mathcal{P}(\mathit{Formulae})$. *Partitions* is the set of all possible partitions.
- $\mathit{instantiate}(\mathit{schema}, \mathit{partition})$ is a procedure that instantiates *schema* with values from *partition*. In particular, for any partition π returned by $\mathit{partitions}(\mathit{schema}, \Gamma)$, $\mathit{instantiate}(\mathit{schema}, \pi) = \Gamma$.

We use the following notational conventions in this chapter:

- $\mathcal{P}(X)$ stands for the power set of a set X.

- For sets X and Y, $X + Y$ stands for the discriminated union $(\{X\} \times X) \cup (\{Y\} \times Y)$. We abuse notation by writing $a \in X + Y$ to mean $(X, a) \in X + Y$ or $(Y, a) \in X + Y$ when the originating set is obvious.

5.2 Basic Definitions

Definition 5.2.1. Result of the search procedure.

The result of the search procedure is one of the symbolic constants *Open*, *Closed* or *Failure*. *Open* and *Closed* have their usual meanings in the context of tableau systems. *Failure* is a third result used to indicate that the search was unable to proceed. For example, a tactic that forces the application of an inapplicable rule will cause a result of *Failure*.

We will also refer to the opposite of a result in our procedures: the opposite of *Open* is *Closed* and the opposite of *Closed* is *Open*. *Failure* has no opposite. Notionally, the opposite of *Failure* is "success": either *Open* or *Closed*, but we do not use this in our procedures.

For convenience, we also define the set $Results = \{Open, Closed, Failure\}$.

Definition 5.2.2. Alternations.

The version of the TWB described in Chapter 4 uses three abstract types of alternation. These alternations appear at different levels of the procedure: the formula choice, rule choice or denominator choice level. In the context of our *Results*, they are:

One-success alternation: Commit to the first choice (formula choice, rule choice, denominator choice) that does not return *Failure* and use its result without considering other alternatives. If all alternatives return *Failure*, return *Failure*.

One-open alternation: If any choice returns *Open*, return *Open*. If all choices return *Failure*, return *Failure*. Otherwise, return *Closed*.

One-closed alternation: If any choice returns *Closed*, return *Closed*. If every alternative returns *Failure*, return *Failure*. Otherwise, return *Open*.

We represent these choices with the symbolic constants *OneSuccess*, *OneOpen* and *OneClosed*. We define the set $Alternatives = \{OneSucces, OneOpen, OneClosed\}$.

Definition 5.2.3. Rules.

A Rule is a 6-tuple: ($Numerator$, $Denominators$, $SideCondition$, $Action$, $FormulaChoiceAlt$, $DenominatorAlt$). Each component of the tuple is as follows:

- $Numerator \in \mathcal{P}(Schemas)$
- $Denominators \in \mathcal{P}(\mathcal{P}(Schemas)) + Results$
- $SideCondition \in Partitions \times Histories \rightarrow \{True, False\}$
- $Action : Partitions \times Histories \rightarrow Histories$

- $FormulaChoiceAlt \in Alternatives$

- $DenominatorAlt \in Alternatives$

We also use the name of each entry in the tuple to define projection functions. For instance, $Numerator(r)$ is a function that extracts the numerator from a rule r. The others are similar.

Definition 5.2.4. Tactics.

We use the basic tactic structure defined in [Martin et al. 1996], but extend the definition to accommodate our additional types of alternation. A tactic is defined by the following grammar:

$$\begin{array}{rcl} \text{tactic} & ::= & Skip \\ & | & Fail \\ & | & Rule(r) \\ & | & \text{tactic}; \text{tactic} \\ & | & \text{tactic!tactic} \\ & | & \text{tactic}|\text{tactic} \\ & | & \text{tactic}||\text{tactic} \\ & | & \mu X.T(X) \end{array}$$

Where r is a rule, X is a variable and $T(X)$ is a tactic in which X may appear as if it were a tactic. We will use ; to sequence tactics, ! for one-success alternation, | for one-open alternation and || for one-closed alternation. $\mu X.T(X)$ represents the least fixpoint of a tactic: each occurence of X within T will be equivalent to $\mu X.T(X)$.

Definition 5.2.5. Tactic Expansion.

We define a function *expand* which re-writes tactics to ensure that the top level of a tactic is either a primitive ($Skip$, $Fail$ or $Rule(r)$), a sequence of the form $Rule(r); t$ or an alternation $t_1!t_2$, $t_1|t_2$ or $t_1||t_2$.

$$\begin{aligned}
expand(Skip) &= Skip \\
expand(Fail) &= Fail \\
expand(Rule(r)) &= Rule(r) \\
expand(Skip; t_1) &= expand(t_1) \\
expand(Fail; t_1) &= Fail \\
expand(Rule(r); t_1) &= Rule(r); t_1 \\
expand((t_1; t_2); t_3) &= expand(t_1; (t_2; t_3)) \\
expand((t_1 ! t_2); t_3) &= expand(t_1; t_3) ! expand(t_2; t_3) \\
expand((t_1 | t_2); t_3) &= expand(t_1; t_3) | expand(t_2; t_3) \\
expand((t_1 || t_2); t_3) &= expand(t_1; t_3) || expand(t_2; t_3) \\
expand((\mu X.T(X)); t_1) &= expand(expand(\mu X.T(X)); t_1) \\
expand(t_1 ! t_2) &= t_1 ! t_2 \\
expand(t_1 | t_2) &= t_1 | t_2 \\
expand(t_1 || t_2) &= t_1 || t_2 \\
expand(\mu X.T(X)) &= expand(T[X := \mu X.T(X)])
\end{aligned}$$

Associativity of ; and distribution of ; over alternation (on the right only) are defined as laws in [Martin et al. 1996]. The alternation of *Angel* is completely nondeterministic. Our alternation operators are essentially specialisations of Angel's generic alternation that handle the trichotomy of results in our system.

5.3 The Search Procedure

5.3.1 Overview

Recall from Section 3.3.5 that there are three types of decisions that the search procedure needs to make as it applies rules:

- Which rule to apply to a set of formulae,
- Which partition of the numerator is used to instantiate denominators, and
- Which denominator to explore next.

These decisions are contained within a set of three mutually-recursive procedures *evalTWB*, *applyRule* and *evalDenominators*. *evalTWB* selects the rule to apply from a given tactic. *applyRule* partitions formulae, checks side conditions and instantiates denominators. *evalDenominators* evaluates a set of denominators and collects an overall result.

5.3.2 Evaluating a Formula Set

The procedure $evalTWB(\Gamma, history, tactic)$ takes a formula set Γ, the current history state and a tactic to evaluate. $evalTWB(\Gamma, history, tactic) = evalTWB'(\Gamma, history, expand(tactic))$, where $evalTWB'$ is defined as follows:

```
$evalTWB'(\Gamma, history, Skip) = Open$
$evalTWB'(\Gamma, history, Fail) = Failure$
$evalTWB'(\Gamma, history, Rule(r)) = applyRule(r, \Gamma, history, Skip)$
$evalTWB'(\Gamma, history, Rule(r); t) = applyRule(r, \Gamma, history, t)$
$evalTWB'(\Gamma, history, t_1!t_2) = $ let $res = evalTWB(\Gamma, history, t_1);$
    if $res == Failure$ then $evalTWB(\Gamma, history, t_2)$ else $res$
$evalTWB'(\Gamma, history, t_1|t_2) = $ let $res_1 = evalTWB(\Gamma, history, t_1);$
    case $res_1$ of {
        $Open \rightarrow Open$
        $Closed \rightarrow$ let $res_2 = evalTWB(\Gamma, history, t_2);$
            if $res_2 == Failure$ then $Closed$ else $res_2$
        $Failure \rightarrow evalTWB(\Gamma, history, t_2)$
    }
$evalTWB'(\Gamma, history, t_1||t_2) = $ let $res_1 = evalTWB(\Gamma, history, t_1);$
    case $res_1$ of {
        $Open \rightarrow$ let $res_2 = evalTWB(\Gamma, history, t_2);$
            if $res_2 == Failure$ then $Open$ else $res_2$
        $Closed \rightarrow Closed$
        $Failure \rightarrow evalTWB(\Gamma, history, t_2)$
    }
```

5.3.3 Rule Application

$applyRule(rule, \Gamma, history, tactic)$ is a procedure that applies a rule to a formula set Γ and a given history, then evaluates *tactic* across the denominator formulae:

```
$applyRule(rule, \Gamma, history, tactic) =$
    **case** $Denominators(rule)$ **of** {
        $Open|Closed|Failure \rightarrow$
            **foreach** $partition$ **in** $partitions(Numerator(rule), \Gamma)$ {
                **if** $SideCondition(rule)(partition, history) == True$
                **then return** $Denominators(rule)$
            }
        **case** $FormulaChoiceAlt(rule)$ **of** {
            $OneOpen \rightarrow applyRuleSelect(Open, rule, \Gamma, history, tactic)$
            $OneClosed \rightarrow applyRuleSelect(Closed, rule, \Gamma, history, tactic)$
            $OneSuccess \rightarrow applyRuleSuccess(rule, \Gamma, history, tactic)$
        }
    }
}
```

Where *applyRuleSelect* and *applyRuleSuccess* are defined as follows:

```
$applyRuleSelect(value, rule, \Gamma, history, tactic) =$
    **let** $res = Failure;$
    **foreach** $partition$ **in** $partitions(Numerator(rule), \Gamma)$ {
        **if** $SideCondition(rule)(partition, history) == True$ **then** {
            **let** $r = evalDenominators(rule, partition, history, tactic);$
            **if** $r = value$ **then return** $value$
            **else if** $r == opposite(value)$ **then** $res = r$
        }
    }
    **return** $res$
```

```
$applyRuleSuccess(rule, \Gamma, history, tactic) =$
    foreach $partition$ in $partitions(Numerator(rule), \Gamma)$ {
        if $SideCondition(rule)(partition, history) == True$ then {
            let $r = evalDenominators(rule, partition, history, tactic);$
            if $r \neq Failure$ then return $r$
        }
    }
    return $Failure$
```

5.3.4 Evaluating Denominators

evalDenominators(*rule*, *partition*, *history*, *tactic*) is a procedure that instantiates the denominators of *rule* according to *partition* and evaluates them using *history* and *tactic*:

```
$evalDenominators(rule, partition, history, tactic) =$
    case $DenominatorAlt(rule)$ of {
        $OneOpen \rightarrow evalDenominatorsSelect(Open, rule, partition, history, tactic)$
        $OneClosed \rightarrow evalDenominatorsSelect(Closed, rule, partition, history, tactic)$
        $OneSuccess \rightarrow evalDenominatorsSuccess(rule, partition, history, tactic)$
    }
```

```
$evalDenominatorsSelect(value, rule, partition, history, tactic) =$
    let $res = Failure;$
    foreach $denominator$ in $Denominators(rule)$ {
        let $instance = instantiate(denominator, partition);$
        let $r = evalTWB(instance, Action(rule)(partition, history), tactic);$
        if $r == value$ then return $value$
        else if $r == opposite(value)$ then $res = r$
    }
    return $res$
```

```
$evalDenominatorsSuccess(rule, partition, history, tactic) =$
    **foreach** $denominator$ **in** $Denominators(rule)$ {
        **let** $instance = instantiate(denominator, partition)$;
        **let** $r = evalTWB(instance, Action(rule)(partition, history), tactic)$;
        **if** $r \neq Failure$ **then return** r
    }
    **return** Failure
```

Chapter 6

Examples and Results

In this chapter, we demonstrate the expressiveness of our semantics by defining four provers for the modal logic K and one for bi-modal K. We also provide a prover for the logic KT, to demonstrate the use of side conditions and actions.

6.1 K

6.1.1 Conventions

We let *Formulae* be the set of K-formulae described in section 3.4.1. We also let *Schemas* be the set of schemas that describe K-formulae. Tableau calculi for K need no history, so we use $\{\emptyset\}$ (the set containing only the empty set) as the set of all possible K-histories. We define a function $noaction : Partitions \times \{\emptyset\} \rightarrow \{\emptyset\}$ by:

$$noaction(partition, \emptyset) = \emptyset$$

Because K does not use side-conditions, we define a function $always : Partitions \times \{\emptyset\} \rightarrow \{True, False\}$ by the following equation: $always(partition, \emptyset) = True$.

Definitions of *partition* and *instantiate* for K are fairly straightforward.

6.1.2 Rules

To be able to use tableau rules, we need to convert them to a 6-tuple encoding. Recall from Section 5.2 that a rule is a 6-tuple (*Numerator*, *Denominators*, *SideCondition*, *Action*, *FormulaChoiceAlt*, *DenominatorAlt*). We walk through the natural encoding of the (Id), $(\vee)$ and (K) rules to illustrate the principles used in the encoding:

$$(Id)\frac{p; \neg p; Z}{\times} \qquad (\vee)\frac{\varphi \vee \psi; Z}{\varphi; Z | \psi; Z} \qquad (K)\frac{\Diamond p; \Box X; Z}{p; X}$$

Figure 6.1: The (Id), $(\vee)$ and (K) rules.

Invertible rules use *OneSuccess* for formula choice alternation and rules that close the tableau have a denominator of $(Results, Closed)$ and ignore the denominator alternation. The (Id) rule therefore has the following encoding:

$$(Id) = (p; \neg p; Z, (Results, Closed), always, noaction, OneSuccess, OneSuccess)$$

Rules that require all denominators to close use *OneOpen* for denominator alternation:

$$(\vee) = (\varphi \vee \psi; Z, (Schemas, \{\{\varphi; Z\}, \{\psi; Z\}\}), always, noaction, OneSuccess, OneOpen)$$

The (K) rule can close the tableau if one of the formula choices close, so *OneClosed* is used for formula choice alternation:

$$(K) = (\Diamond\varphi; \Box X; Z, (Schemas, \{\{\varphi; X\}\}), always, noaction, OneClosed, OneSuccess)$$

The remaining rules of our tableau calculus for K have a straightforward encoding:

$$(\bot) = (\bot; Z, (Results, Closed), always, noaction, OneSuccess, OneSuccess)$$
$$(\wedge) = (\varphi \wedge \psi; Z, (Schemas, \{\{\varphi; \psi; Z\}\}), always, noaction, OneSuccess, OneSuccess)$$

Our first prover for K is straightforward. We apply the $(\bot)$, (Id), $(\wedge)$ and $(\vee)$ rules as much as possible. If this does not give us a closed tableau, we attempt to apply the (K) rule and continue:

$$\mathbb{K}_1 = \mu X.((Rule(\bot)!(Rule(Id)!(Rule(\wedge)!(Rule(\vee)!Rule(K))))); X)!Skip$$

For readability, we make the following abuses of notation:

- We use the name of a rule (ρ) to stand for $Rule(\rho)$.
- We omit the parentheses around a rule.
- We assume that all alternations are right-associative.

This gives us the following simplified version of $\mathbb{K}_1$:

$$\mathbb{K}_1 = \mu X.((\bot!Id!\wedge!\vee!K); X)!Skip$$

The choice of alternation operators is important. For alternations across invertible rules the logical result is not changed but the amount of backtracking can vary greatly. As we have seen in Chapter 3, the order of rule applications does not matter for invertible rules: if we can apply a rule, we can commit to the application without worry. If we use one-closed alternation between rules, we get the prover $\mathbb{K}_{closed}$:

$$\mathbb{K}_{closed} = \mu X.((\bot||Id|| \wedge || \vee ||K); X)!Skip$$

If the input has a closed tableau, then no redundant backtracking takes place. If the input does not have a closed tableau then $\mathbb{K}_{closed}$ will try every possible combination of the $(\bot)$, (Id), $(\wedge)$ and $(\vee)$ rules looking for a closed tableau that does not exist.

If we instead use one-open alternation, we get an unsound prover:

$$\mathbb{K}_{open} = \mu X.((\bot|Id|\wedge|\vee|K);X)!Skip$$

Suppose we give $\mathbb{K}_{open}$ the input set $\{\bot, \Diamond p\}$. Our tactic will choose to apply $(\bot)$, generating a closed result. The one-open alternation means that it will not return this result, but continue trying alternative rules until it comes to the (K) rule. Applying the (K) rule then generates the denominator $\{p\}$, which has an open result. The one-open alternation then returns open as the result of the search, which is incorrect. Note that the problem does not invalidate our result about committing to invertible rules. The unsoundness comes from how the tactic combinator for one-open alternation collects results. It is simply the incorrect choice for this situation.

We will benchmark $\mathbb{K}_{closed}$ along with our other provers, but for the remaining provers we will only use one-success alternation between invertible rules.

We can lift the alternation from the denominator of the $(\vee)$ rule and move it into the tactics by using the following pair of projective rules:

$$(\vee_1)\frac{\varphi \vee \psi; Z}{\varphi; Z} \qquad\qquad (\vee_2)\frac{\varphi \vee \psi; Z}{\psi; Z}$$

These rules have the following encoding:

$$(\vee_1) = (\varphi \vee \psi; Z, (Schemas, \{\{\varphi; Z\}\}), always, noaction, OneSuccess, OneSuccess)$$
$$(\vee_2) = (\varphi \vee \psi; Z, (Schemas, \{\{\psi; Z\}\}), always, noaction, OneSuccess, OneSuccess)$$

We define another prover, $\mathbb{K}_2$, using the following tactic:

$$\mathbb{K}_2 = \mu X.((\bot!Id!\wedge!(\vee_1|\vee_2)!K);X)!Skip$$

We can also encode the recursive version of the (K) rule:

$$(K')\frac{\Diamond\varphi; \Box X; \Diamond Y; Z}{\varphi; X || \Diamond Y; \Box X}$$

$$(K') = (\Diamond\varphi; \Box X; \Diamond Y; Z, (Schemas, \{\{\varphi; X\}, \{\Box X; \Diamond Y\}\}), always, noaction, OneSuccess, OneClosed)$$

This replaces one-closed formula alternation with one-closed denominator alternation. Our prover $\mathbb{K}_3$ uses the same type of tactic as $\mathbb{K}_1$:

$$\mathbb{K}_3 = \mu X.((\bot!Id!\wedge!\vee!K');X)!Skip$$

We can also use the projective-or rules $(\vee_1)$ and $(\vee_2)$ with our recursive-K rule, yielding the prover $\mathbb{K}_4$:

$$\mathbb{K}_4 = \mu X.((\bot!Id!\wedge!(\vee_1|\vee_2)!K'); X)!Skip$$

6.2 Bi-modal K

Recall from section 3.5 that bi-modal K is essectially the same as K with an additional $(K\blacklozenge)$ rule:

$$(K\blacklozenge)\frac{\blacklozenge\varphi; \blacksquare X; Z}{\varphi; X}$$

$$(K\blacklozenge) = (\blacklozenge\varphi; \blacksquare X; Z, (Schemas, \{\{\varphi; X\}\}), always, noaction, OneClosed, OneSuccess)$$

With appropriate (minor) changes to the definitions of *Formulae* and *Schemas*, we can construct a prover for bi-modal K. As with the projective-or rules given previously, our prover may need to backtrack and select an alternate rule if we do not get a satisfactory result. The prover $\mathbb{K}_2$ uses one-open alternation as both the application of $(\vee_1)$ and $(\vee_2)$ need to generate a closed result to produce an overall closed result. It is sufficient for one of either $(K\Diamond)$ or $(K\blacklozenge)$ to generate a closed result, so we use one-closed alternation in the tactic:

$$\mathbb{K}\Diamond\blacklozenge = \mu X.(\bot!Id!\wedge!\vee!(K\Diamond||K\blacklozenge); X)!Skip$$

6.3 KT

As discussed in section 3.5.3, the construction of a terminating procedure for KT requires the use of histories and side conditions to prevent infinite applications of the (T) rule.

Our definitions of *Formulae* and *Schemas* are unchanged from those used for K, but we let $Histories = \mathcal{P}(Formulae)$. We define functions for manipulating histories:

$$\begin{aligned}
add &: Partitions \times Histories \rightarrow Histories \\
add(partition, history) &= history \cup partition(\varphi) \\
clear &: Partitions \times Histories \rightarrow Histories \\
clear(partition, history) &= \emptyset \\
noaction &: Partitions \times Histories \rightarrow Histories \\
noaction(partition, history) &= history
\end{aligned}$$

We also define the following functions for use as side conditions:

$$\begin{aligned} always &: Partitions \times Histories \rightarrow \{True, False\} \\ always(partition, history) &= True \\ notin &: Partitions \times Histories \rightarrow \{True, False\} \\ notin(partition, history) &= \begin{cases} True & \text{if } partition(\varphi) \not\subset history \\ False & \text{otherwise} \end{cases} \end{aligned}$$

We can then define the $(\bot), (Id), (\wedge)$ and $(\vee)$ rules as before, using our KT-specific versions of *always* and *noaction* for side conditions and actions.

The (T) rule is only allowed to apply if φ is not in the history of unboxed formulae. Once applied, the newly-unboxed formula is added to the history:

$$(T)\frac{\Box\varphi}{\varphi;\Box\varphi \quad - \quad history : history \cup \{\varphi\}}(\varphi \notin history)$$

$$(T) = (\Box\varphi, (Schemas, \{\{\varphi, \Box\varphi\}\}), notin, add, OneSuccess, OneSuccess)$$

Notice that instead of *noaction*, we use the *add* function to add whatever is in φ to the history.

The (K) rule is essentially unchanged, but we need to clear the list of unboxed formulae when it is applied:

$$(K) = (\Diamond\varphi; \Box X; Z, (Schemas, \{\{\varphi; X\}\}), always, clear, OneClosed, OneSuccess)$$

This gives us the following prover for KT:

$$\mathbb{KT} = \mu X.((\bot!Id!\wedge!\vee!T!K); X)!Skip$$

6.4 Limitations

Our revised semantics are sufficiently expressive to emulate the basic features of the TWB, but the TWB has some advanced features which we have not attempted to replicate:

Upward Variables: The TWB allows the definition of additional variables which are propagated from the leaves of a tableau towards the root. The status of a node in the tableau (open or closed) is essentially a special case of upward variables. Variables are local to a node in the search but a rule can access the rules of its immediate children through the `BACKTRACK` directive.

Backtracking actions: A `BACKTRACK` directive can be attached to a rule to execute arbitrary O'Caml code before the search returns to its parent.

Custom branch conditions: The TWB provides a `BRANCH` directive that can be attached to a rule. `BRANCH` specifies a condition in a similar fashion to `COND`. The

BRANCH condition checked whenever the search returns to a node to decide if the search should proceed with the next denominator or return to the parent rule application.

Conditional Branching: An additional branching operator ||| is provided. This operator makes no assumptions about when to cease exploring child nodes or how to combine results of an alternation. The user is expected to provide the branch condition and a method of determining the node's status by making use of the BRANCH and BACKTRACK directives.

Extended Denominators: The TWB extends the concept of denominators by allowing the user to specify history variables, upward variables and functions that return formulae. The following is an example from [Abate 2007b] which implements semantic braching for the $(\vee)$ rule:

```
RULE Or
       { A v B }
  =====================
  A | B ; nnf_term (~ A)
END
```

nnf_term is an O'Caml function defined alongside the tableau that converts a formula into its negation normal form. An example of a rule that uses history variables in its denominator is the *S*4 rule presented in [Abate and Goré 2009]:

```
RULE S4
  { <> P } ; Z
  -------------
    P ; UBXS
  COND notin(<> P, DIAS)
  ACTION [ DIAS := add(<> P, DIAS) ]
END
```

UBXS is the name of a history variable that was introduced by the TWB's standard HISTORIES declaration. The contents of UBXS are copied into the new formula set whenever the S4 rule is applied.

6.5 Experimental Results

We have implemented the system described in Chapter 5 as an O'Caml library called "twbcore". Twbcore is mostly implemented as an O'Caml functor called MkTableau. A functor is a module that is parameterised by another module. MkTableau is passed a module that names the types used for formulae and histories. The module parameter is also required to provide an implementation of a unification procedure and an instantiation procedure to instantiate formulae from a unifier. Together, this is sufficient for twbcore to define a logic-specific type for tableau rules and an implementation of our search procedure that is specialised for that particular logic. Twbcore also provides a polymorphic type for tactics and substitution and expansion operations on tactics that match the description given in Section 5.2. Twbcore is as close to a direct translation of the procedures from Chapter 5 as possible. We do not attempt any "tricks" to improve run-time performance.

We then used twbcore to implement the provers for the modal logic K from Section 6.1: $\mathbb{K}_{closed}$, $\mathbb{K}_1$, $\mathbb{K}_2$, $\mathbb{K}_3$ and $\mathbb{K}_4$ and the prover $\mathbb{KT}$ from Section 6.3.

In this section, we present and discuss the results of benchmarking our provers for the logics K and KT. We use the standard benchmark suite provided by the authors of the Logic Work Bench. The suite is descibed in [Heuerding and Schwendimann 1996].

6.5.1 The LWB benchmarks

In [Heuerding and Schwendimann 1996] three sets of benchmark formulae are presented: one each for the logics K, KT and S4. Each benchmark consists of 9 classes of provable formulae and 9 classes of unprovable formulae. Within each class, there are 21 formulae of increasing complexity but similar basic structure.

The benchmark method is also prescribed by [Heuerding and Schwendimann 1996]. Each prover being benchmarked is run against the formulae of each class in increasing order of complexity. We run the provers with a timeout of 100 seconds and note the most complex formula from each class that was successfully decided. We also record the running times for each formula.

We also present the basic details of our twbcore system in the format specified by [Heuerding and Schwendimann 1996]:

Prover: We used provers built from the still-in-development twbcore library, version 0.42. Twbcore provides a logic-agnostic tableau search procedure which we combine with specific tableau rules and a strategy to construct a complete prover. Twbcore is written in O'Caml and compiled with `ocamlc` version 3.10.0 running on Ubuntu 8.04 LTS (x86_64).

Availability: Twbcore has not yet been released. The sources for the provers given here will be included as examples in the source distribution.

Additional Facilities: Twbcore is intended to be the core of a logic-agnostic toolkit for building tableau-based provers. As such, it favours flexibility over speed. Twbcore is single-threaded.

Hardware: All tests were conducted on a Pentium Dual-Core E5200 running at 2.5GHz with 2GB of RAM.

Timing Method: An external driver script written in Python that extracts the formula from the input file and feeds it to the prover. We negated the input formulae in a separate preprocessing step. Provable formulae will then generate a closed result and non-provable formulae an open result. The driver script also measured the running time of the program by computing difference in wall-clock time. The timeout was enforced by setting up a `SIGALRM` signal to be sent to the driver after the 100 seconds had expired. The driver script is provided in Appendix A

We summarise the collected data here, but full tables of collected data are presented in Appendix A.

6.5.2 K

Recall our conjecture from Section 6.1.2 that formulae with an open tableau will cause large amounts of redundant backtracking for $\mathbb{K}_{closed}$, while formuale with a closed tableau will not have redundant backtracking. Our results show the conjecture to be correct. The unprovable formulae in Figure 6.2 are designed to have open tableau (after negation) and we see that $\mathbb{K}_{closed}$ is significantly less capable when compared to the other provers that use one-success alternation between rules. We also notice that the four variants $\mathbb{K}_1$ through $\mathbb{K}_4$ are approximately equally capable: This is not a surprising result but it is encouraging. We would expect alternation to have approximately the same cost regardless of its level in the search procedure (rule-choice, formula-choice or denominator-choice). We do not want future end users to be pressured into selecting a particular level of alternation based on real or imagined efficiency benefits.

	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
k_branch_n	0	1	2	3	2
k_d4_n	3	4	4	4	4
k_dum_n	4	18	18	17	17
k_grz_n	2	8	8	8	8
k_lin_n	2	3	3	3	3
k_path_n	2	6	5	6	5
k_ph_n	2	3	3	3	3
k_poly_n	2	18	18	18	18
k_t4p_n	1	7	7	6	6

Figure 6.2: Hardest solvable unprovable K formulae (by class and prover).

For provable K formulae, we see some interesting results. $\mathbb{K}_{closed}$ is slightly less capable than the other $\mathbb{K}$ systems. We conjecture that this is a lesser version of the effect observed with unprovable formulae: If the $\mathbb{K}_{closed}$ prover chooses to explore a branch that turns out to be open, it will perform some redundant backtracking within that branch to confirm that the branch is indeed open. As the whole tableau closes, the scope for redundant backtracking is restricted.

The other interesting result is the behaviour of $\mathbb{K}_3$ and $\mathbb{K}_4$ on the k_poly_p set. The only difference between these provers and $\mathbb{K}_1$ and $\mathbb{K}_2$ use the recursive version of the (K) rule that can commit to its choice of $\Diamond$-formulae. It would therefore appear that our mechanism for backtracking over formula choice is significantly less efficient than our other forms of backtracking.

	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
k_branch_p	1	1	1	1	1
k_d4_p	5	5	5	5	5
k_dum_p	4	7	7	7	7
k_grz_p	4	6	6	6	6
k_lin_p	5	5	5	5	5
k_path_p	2	2	2	2	2
k_ph_p	2	2	2	2	2
k_poly_p	5	5	5	13	13
k_t4p_p	2	6	6	5	5

Figure 6.3: Hardest solvable provable K formulae (by class and prover).

6.5.3 KT

Our implementation of $\mathbb{KT}$ was not intended to be a serious prover but rather a proof of concept, and this shows in the complexity of formulae it was able to evaluate within the 100 second time limit (Figure 6.4). Our prover used O'Caml's built-in linked-list data type to implement the history of examined $\Box$-formulae (see Section 3.5.3), which has an $O(n)$ membership test. The membership test is exectued every time the prover attempts to apply the (T) rule. This means that the "saturation step", where the prover applies the $(\bot)$, (Id), $(\wedge)$, $(\vee)$ and (T) rules as much as possible, is $O(n^2)$ in the number of $\Box$-formulae. The functional set data structure provided by the O'Caml standard library has $O(\log n)$ membership testing and insertion, which would reduce the cost of saturating $\Box$-formulae to $O(n \log n)$ in the number of $\Box$-formulae. (Of course, the saturation step is still $O(2^n)$ in the number of $\vee$-connectives in the formula set). With such low numbers of solved formulae in each class, there is

Class name	Hardest formula
kt_45_n	1
kt_branch_n	3
kt_dum_n	3
kt_grz_n	> 20
kt_md_n	4
kt_path_n	2
kt_ph_n	3
kt_poly_n	1
kt_t4p_n	2

(a) Unprovable formulae

Class name	Hardest formula
kt_45_p	3
kt_branch_p	2
kt_dum_p	1
kt_grz_p	0
kt_md_p	4
kt_path_p	1
kt_ph_p	2
kt_poly_p	1
kt_t4p_p	1

(b) Provable formulae

Figure 6.4: Hardest solvable KT formulae (by class).

not enough data here to make any reasonable inferences about the scaling of our $\mathbb{KT}$

prover. Since so many classes of formulae scaled so poorly, we do not attempt to draw any conclusions about the prover and recognise it for what is: a proof of concept.

Chapter 7

Conclusion

We provide a short summary of the contributions of this thesis:

- From our study of the TWB have identified what we believe to be the essential features of a generic tableau-driven prover.
- We have identified multiple shortcomings with the specification of the TWB and provided a high-level, informal specification that encompasses just these essential features.
- We have shown that our system is sufficiently expressive to permit the implementation of some simple modal logics with distinct backtracking and history-checking properties, and that it allows significant flexibility in how the user chooses to express them.
- We have tested our provers against the Logic Work Bench's standard benchmark suites and examined how different tactics can significantly impact the running time of a prover or even render it logically unsound.

7.1 Further Work

We outline multiple areas where our work could fruitfully be extended:

- Test the expressive power of our system further, by attempting to implement more complex logics.
- Our semantics describe how to compute the result of trying to build a tableau of a formula set and our sample implementation is essentially a direct translation of this. It would be useful for our implementation to optionally store the tableau as it is explored, to enable the generation of a witness that shows why the computed result is correct.
- Decouple rule application from the rest of the search procedure and allow it to be used standalone as a generic tool to manually inspect tableau systems.

- Extend the front-end parser of the current TWB to support the three types of alternation (one-success, one-open and one-closed) at all levels and replace the backend with our implementation.
- Using our pseudocode as a starting point, formalise the semantics of our generic, depth-first tableau search.
- Extend the semantics presented in Chapter 5 to incorporate the advanced features of the TWB described in Section 6.4.
- Improve performance of the implementation. Although speed was not our primary concern, a revised system must be fast enough for day-to-day usage to be a viable replacement.
- Extend the procedure to support multi-pass decision procedures, something that the TWB cannot yet do.

Appendix A

Running Times and Driver Script

This chapter contains the running time data collected for the benchmarks in Section 6.5, along with a listing of the driver script used to generate them. All running times are in seconds. We do not bother collecting running times beyond the first 100-second timeout, which we indicate with a $-$.

A.1 K

A.1.1 Unprovable Formulae

A.1.1.1 k_branch_n

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	-	0.07	0.01	0.07	0.10
2		-	1.55	1.03	1.55
3			-	78.4	-

A.1.1.2 k_d4_n

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	0.01	0.00	0.00	0.00	0.00
2	0.75	0.03	0.05	0.04	0.04
3	29.05	0.51	0.60	0.60	0.71
4	-	9.18	11.40	11.59	14.36
5		-	-	-	-

A.1.1.3 k_dum_n

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	0.61	0.04	0.05	0.05	0.05
2	3.24	0.05	0.05	0.05	0.06
3	3.03	0.04	0.05	0.05	0.06
4	19.27	0.05	0.05	0.05	0.08
5	-	0.05	0.06	0.06	0.07
6		0.07	0.08	0.08	0.08
7		0.09	0.10	0.10	0.11
8		0.14	0.15	0.16	0.17
9		0.23	0.25	0.26	0.28
10		0.42	0.44	0.48	0.50
11		0.77	0.81	0.87	0.92
12		1.49	1.54	1.69	1.77
13		2.87	2.99	3.29	3.42
14		5.69	5.91	6.45	6.93
15		11.17	11.58	12.73	13.38
16		22.28	23.06	25.40	26.78
17		44.20	46.04	51.52	53.75
18		89.47	93.67	-	-
19		-	-		

A.1.1.4 k_grz_n

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	1.24	0.07	0.09	0.09	0.10
2	22.85	0.16	0.20	0.18	0.20
3	-	0.23	0.28	0.29	0.33
4		0.08	0.10	0.10	0.11
5		0.40	0.48	0.50	0.59
6		0.12	0.13	0.15	0.17
7		10.54	13.95	13.70	16.29
8		0.12	0.14	0.15	0.16
9		-	-	-	-

A.1.1.5 k_lin_n

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	0.00	0.00	0.00	0.00	0.00
2	24.0	0.05	0.06	0.05	0.06
3	-	1.34	1.70	1.50	1.75
4		-	-	-	-

A.1.1.6 k_path_n

[h]

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	0.24	0.07	0.09	0.07	0.08
2	16.92	0.77	0.91	0.78	0.86
3	-	2.94	3.61	2.98	3.35
4		12.03	14.94	12.18	13.93
5		30.92	39.23	31.20	35.09
6		95.89	-	99.25	-
7		-		-	

A.1.1.7 k_ph_n

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	0.00	0.00	0.01	0.00	0.00
2	1.59	0.01	0.01	0.01	0.01
3	-	4.45	6.74	4.51	6.09
4		-	-	-	-

A.1.1.8 k_poly_n

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	0.04	0.01	0.01	0.01	0.01
2	0.48	0.02	0.02	0.02	0.02
3	-	0.08	0.08	0.08	0.08
4		0.14	0.15	0.14	0.15
5		0.39	0.41	0.38	0.40
6		0.61	0.66	0.58	0.63
7		1.37	1.46	1.28	1.37
8		1.95	2.07	1.83	1.97
9		3.76	4.02	3.48	3.73
10		5.07	5.46	4.69	5.07
11		8.94	9.57	8.19	8.80
12		11.55	12.35	10.65	11.42
13		18.86	20.25	17.25	18.66
14		23.71	25.57	21.71	23.34
15		36.66	39.24	33.34	35.76
16		44.88	48.39	40.88	43.80
17		66.05	70.85	60.59	64.62
18		79.60	84.70	72.66	77.46
19		-	-	-	-

A.1.1.9 k_t4p_n

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	0.80	0.01	0.01	0.01	0.01
2	-	0.08	0.08	0.08	0.09
3		0.29	0.31	0.38	0.41
4		1.21	1.29	1.64	1.77
5		4.83	5.17	6.95	7.50
6		19.53	20.67	30.23	32.26
7		78.42	82.55	-	-
8		-	-		

A.1.2 Provable Formulae

A.1.2.1 k_branch_p

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	39.62	0.96	1.44	0.97	1.44
2	-	-	-	-	-

A.1.2.2 k_d4_p

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	0.00	0.01	0.01	0.00	0.01
2	0.07	0.07	0.09	0.07	0.09
3	0.35	0.33	0.42	0.35	0.44
4	7.77	5.10	6.57	5.13	6.49
5	61.65	32.19	40.61	43.93	54.89
6	-	-	-	-	-

A.1.2.3 k_dum_p

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	0.07	0.03	0.04	0.03	0.04
2	0.86	0.19	0.22	0.20	0.24
3	0.86	0.19	0.22	0.20	0.24
4	11.09	0.94	1.12	1.02	1.22
5	-	3.28	4.03	3.49	4.26
6		11.77	14.49	13.00	15.89
7		32.53	40.43	35.46	43.77
8		-	-	-	-

A.1.2.4 k_grz_p

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	0.22	0.09	0.10	0.11	0.13
2	0.54	0.18	0.21	0.21	0.25
3	1.53	0.52	0.61	0.62	0.72
4	14.18	2.37	2.75	2.80	3.24
5	-	11.52	13.50	14.45	16.69
6		69.03	80.49	83.20	96.51
7		-	-	-	-

A.1.2.5 k_lin_p

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	0.00	0.00	0.00	0.00	0.00
2	0.01	0.01	0.01	0.01	0.01
3	0.10	0.10	0.12	0.10	0.12
4	0.96	0.96	1.15	1.01	1.20
5	8.34	8.33	9.98	8.77	10.51
6	-	-	-	-	-

A.1.2.6 k_path_p

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	0.00	0.00	0.00	0.00	0.00
2	4.28	4.21	5.55	4.69	6.06
3	-	-	-	-	-

A.1.2.7 k_ph_p

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	0.00	0.00	0.00	0.00	0.00
2	0.04	0.05	0.07	0.46	0.07
3	-	-	-	-	-

A.1.2.8 k_poly_p

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	0.01	0.01	0.01	0.01	0.01
2	0.17	0.17	0.22	0.06	0.06
3	0.69	0.69	0.93	0.14	0.14
4	8.76	8.75	12.72	0.57	0.57
5	28.88	28.80	43.14	1.02	1.02
6	-	-	-	2.89	2.89
7				4.56	4.56
8				10.42	10.42
9				15.16	15.16
10				30.31	30.31
11				41.75	41.75
12				75.68	75.68
13				99.83	99.83
14				-	-

A.1.2.9 k_t4p_p

Formula Number	$\mathbb{K}_{closed}$	$\mathbb{K}_1$	$\mathbb{K}_2$	$\mathbb{K}_3$	$\mathbb{K}_4$
1	0.21	0.12	0.13	0.13	0.15
2	17.63	0.87	1.01	1.03	1.20
3	-	4.58	5.34	5.09	5.88
4		11.53	13.38	21.17	24.45
5		27.77	32.32	74.28	85.62
6		71.11	82.41	-	-
7		-	-		

A.2 KT

A.2.1 Unprovable Formulae

A.2.1.1 kt_45_n

Formula Number	Time (seconds)
1	0.65
2	-

A.2.1.2 kt_branch_n

Formula Number	Time (seconds)
1	0.08
2	1.09
3	81.43
4	-

A.2.1.3 kt_dum_n

Formula Number	Time (seconds)
1	0.10
2	72.81
3	76.39
4	-

A.2.1.4 kt_grz_n

Formula Number	Time (seconds)
1	0.01
2	0.01
3	0.01
4	0.01
5	0.02
6	0.02
7	0.02
8	0.03
9	0.04
10	0.05
11	0.06
12	0.07
13	0.08
14	0.10
15	0.11
16	0.13
17	0.16
18	0.19
19	0.26
20	0.27
21	0.30

A.2.1.5 kt_md_n

Formula Number	Time (seconds)
1	0.00
2	0.00
3	0.00
4	0.29
5	-

A.2.1.6 kt_path_n

Formula Number	Time (seconds)
1	0.34
2	27.05
3	-

A.2.1.7 kt_ph_n

Formula Number	Time (seconds)
1	0.00
2	0.01
3	0.84
4	-

A.2.1.8 kt_poly_n

Formula Number	Time (seconds)
1	14.91
2	-

A.2.1.9 kt_t4p_n

Formula Number	Time (seconds)
1	0.03
2	16.45
3	-

A.2.2 Provable Formulae

A.2.2.1 kt_45_p

Formula Number	Time (seconds)
1	0.19
2	4.92
3	82.07
4	-

A.2.2.2 kt_branch_p

Formula Number	Time (seconds)
1	0.07
2	53.72
3	-

A.2.2.3 kt_dum_p

Formula Number	Time (seconds)
1	44.29
2	-

A.2.2.4 kt_grz_p

Formula Number	Time (seconds)
1	-

A.2.2.5 kt_md_p

Formula Number	Time (seconds)
1	0.00
2	0.00
3	0.02
4	3.34
5	-

A.2.2.6 kt_path_p

Formula Number	Time (seconds)
1	0.00
2	-

A.2.2.7 kt_ph_p

Formula Number	Time (seconds)
1	0.00
2	0.03
3	-

A.2.2.8 kt_poly_p

Formula Number	Time (seconds)
1	1.26
2	-

A.2.2.9 kt_t4p_p

Formula Number	Time (seconds)
1	5.44
2	-

A.3 Driver Script

The following listing shows the driver that we used to time our provers. We expect it to be comprehensible by anyone with experience programming in Python. Note that the `signal` module does not work on Windows systems.

```
#!/usr/bin/python

import os
import signal
import subprocess
import sys
import time

timeout = 100
_, prover, fname = sys.argv
lines = open(fname, 'r').readlines()

class Alarm(Exception):
    pass
def alarm_thrower(signum, frams):
    raise Alarm
signal.signal(signal.SIGALRM, alarm_thrower)

print "testing %s with %s" % (fname,prover)
num = 1
for line in lines:
    child = subprocess.Popen(['/usr/bin/ocamlrun', prover],
                             stdin=subprocess.PIPE,
                             stdout=subprocess.PIPE,
                             stderr=subprocess.STDOUT)
    try:
        signal.alarm(timeout)
        t0 = time.time()
        output = child.communicate(line)[0]
        print output
        signal.alarm(0)
        t1 = time.time()
        dt = t1 - t0
        print "%d: time: %f" % (num, dt)
        num += 1
    except Alarm:
        print "%d: timeout." % num
        os.kill(child.pid, signal.SIGKILL)
        break
```

Bibliography

ABATE, P. 2007a. Tableau work bench (source code). `http://twb.rsise.anu.edu.au/cgi-bin/darcsweb/darcsweb.cgi?r=twb-3.5;a=tree`. (pp. 26, 27)

ABATE, P. 2007b. *The Tableau Workbench: A framework for building automated tableau-based theorem provers*. PhD thesis, Australian National University. (pp. 1, 3, 4, 42)

ABATE, P. AND GORÉ, R. 2003. The tableaux work bench. In M. C. MAYER AND F. PIRRI Eds., *TABLEAUX*, Volume 2796 of *Lecture Notes in Computer Science* (2003), pp. 230–236. Springer. (p. 3)

ABATE, P. AND GORÉ, R. 2009. The tableau workbench. *Electronic Notes in Theoretical Computer Science 231*, 55–67. (pp. 4, 23, 24, 25, 42)

BELNAP, N. 1976. How a computer should think. In *Proceedings of the Oxford International Symposium on Contemporary Aspects of Philosophy* (1976), pp. 30–56. Oxford, England. (p. 1)

BERRE, D. L. AND PARRAIN, A. 2008. On sat technologies for dependency management and beyond. In *SPLC (2)* (2008), pp. 197–200. (p. 1)

CASTILHO, M., FARIÑAS DEL CERRO, L., GASQUET, O., AND HERZIG, A. 1997. Modal tableaux with propagation rules and structural rules. *Fundamenta Informaticae 32*, 3/4. (p. 5)

D'AGOSTINO, M., GABBAY, D. M., HÄHNLE, R., AND POSEGGA, J. Eds. 1999. *Handbook of Tableau Methods*, Chapter Tableau methods for modal and temporal logics, pp. 297–396. Springer. (p. 9)

FARIAS 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 (18-23 juin 2001), pp. 6. Springer Verlag. (pp. 1, 5)

GASQUET, O., SAID, B., AND SCHWARZENTRUBER, F. 2009. A semantics for an event based generic tableau prover. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX), Oslo, Norway, 06/07/2009-10/07/2009* (http://www.uio.no/, 2009). University of Oslo. (pp. 1, 3, 6)

GORÉ, R. AND NGUYEN, L. D. 2000. Cardkt: Automated multi-modal deduction on java cards for multi-application security. In *Java Card Workshop* (2000), pp. 38–51. (p. 1)

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2005. *The Java(TM) Language Specification* (third ed.). Addison-Wesley Professional. (p. 29)

HEUERDING, A., JÄGER, G., SCHWENDIMANN, S., AND SEYFRIED, M. 1996. The logics workbench lwb: a snapshot. *Euromath Bulletin 2*, 1, 177–186. (p. 3)

HEUERDING, A. AND SCHWENDIMANN, S. 1996. A benchmark method for the propositional modal logics k, kt, s4. `http://www.iam.unibe.ch/~lwb/benchmarks/benchmarks.html`. (pp. 3, 6, 43)

LEROY, X., DOLIGEZ, D., GARRIGUE, J., RÉMY, D., AND VOUILLON, J. 2008. *The Objective Caml System, Release 3.11*. Institut National de Recherche en Informatique et en Automatique. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`. (p. 29)

MARTIN, A. P., GARDINER, P. H. B., AND WOODCOCK, J. C. P. 1996. A tactic calculus - full version. *Formal Aspects of Computing 8(E)*, 224–285. `http://www.dcs.glasgow.ac.uk/~jon/facs/e-papers/FACj_8E_pp244-285.ps.Z`. (pp. ix, 4, 24, 31, 32)

MCMILLAN, K. L. 2000. *The SMV System*. Carnegie Mellon University. `http://www.cs.cmu.edu/~modelcheck/smv/smvmanual.ps`. (p. 1)

RUSSELL, S. J. AND NORVIG, P. 2003. *Artificial Intelligence: A Modern Approach*. Pearson Education. (p. 9)

SAHADE, M. 2004. The conditions and actions in lotrec's language. Technical report, Institut de Recherche en Informatique de Toulouse. (p. 5)

STROUSTRUP, B. 1991. *The C++ programming language (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (p. 29)