2013

TR-2013005: Realization Implemented

Melvin Fitting

Realization Implemented

Melvin Fitting
e-mail: melvin.fitting@lehman.cuny.edu
web page: comet.lehman.cuny.edu/fitting

April 2, 2013, revised May 8, 2013

Abstract

Justification logics are connected to modal logics via *realization theorems*. These have both constructive and non-constructive proofs. In this report we do two things. First we provide a new path to constructive realization proofs, going through an intermediate *quasi-realization* stage. Quasi-realizers are easier to produce than realizers, though like them they are constructed from cut-free proofs. Quasi-realizers in turn constructively convert to realizers, and this conversion is independent of the justification logic in question. The construction depends only on the structure of the formula involved.

The second thing we do is provide a Prolog implementation of quasi-realization, and quasi-realization to realization conversion, for the logic LP. Many other justification logics can obviously be treated similarly.

Our quasi-realization algorithm, and its implementation, assumes the underlying modal proof system (for S4) is based on tableaus. Since these may not be familiar to everybody, we provide a sketch of how tableaus work. Then we present our algorithms, our implementation, and a discussion of implementation behavior and design decisions.

We believe our algorithms are simple and straightforward. The original realization algorithm, for instance, needed the entire cut-free proof as input. Our quasi-realization algorithm works one formal proof step at a time. There is, in the literature, another realization construction that works one step at a time, but it requires extensive use of substitution while our quasi-realization algorithm does not. The conversion algorithm is, as noted above, independent of particular justification logics and so only needs to be understood once. It is only here that substitution is needed.

The reason for the length of this report has less to do with algorithm complexity than with the desire to supply background and discussion. We hope we have been able to make the ideas clear.

A text version of the Prolog program discussed here can be obtained from my web site:
`comet.lehman.cuny.edu/fitting`

Contents

List of Figures

List of Tables

Chapter 1

Justification Logics

1.1 Introduction

Justification logics are logics similar to modal logics, but with modal operators replaced by an infinite family of *justifications* that are intended to stand for reasons for things. The first justification logic was introduced by Artemov, [1], and was called LP, for *logic of proofs*. It played an essential role in Artemov's creation of an arithmetic completeness theorem for intuitionistic logic, finishing a line of research that began with Gödel, [15]. As an essential step in that work, LP was shown to have a direct connection with the modal logic S4, via what is called a *Realization Theorem*. This says that every S4 theorem has a realization, a replacement of modal operators by justification terms, that is a theorem of LP. In a sense, such a realization represents the flow of information hidden in the modal operators. Today one often sees references to S4 as a logic in which modal operators *implicitly* represent knowledge, while LP justification terms *explicitly* represent it.

Since Artemov's original work many other justification logics have been created, each connected with a corresponding modal logic via a realization theorem. The phenomenon seems to be much more general than its origins might suggest. It is not known which modal logics have justification counterparts and which do not. Indeed the question itself is a bit fuzzy, since justification logics can differ in the family of operations allowed on justification terms and one might discover an interesting new operation. That is, there is no exact definition of justification logic. So far most work has gone into investigation of justification logics known to correspond to the most familiar of modal logics, K, T, K4, S5, and so on. For a discussion of the evolution into a family of logics of explicit knowledge, see [2, 12]. Indeed, more recently the work has begin to be extended to admit quantification, but this is another story, see [3, 8].

The first proof of a realization theorem is in [1] (though it predates that publication). It is constructive, and makes use of cut-free sequent calculus modal proofs. There have been non-constructive proofs. There is one based on a semantics for justification logics, in [11]. More recently there is one using the model existence theorem, in [14]. But most proofs have been constructive. Typically, cut-free Gentzen sequent proofs play an essential role, but recently prefixed tableaus/nested sequents have been used constructively, to provide a modular approach applicable to a basic family of modal logics, [16].

In this report we will first give a new constructive proof of realization. We present it for LP and S4, but the argument is much more general. Considering one case in detail should be enough to allow for easy extensions. We make use of semantic tableaus rather than a sequent calculus, but there is a close and well-known connection between them. Since tableaus might be less familiar to some readers, we provide a basic introduction to them, sufficient to our purposes.

Our realization approach divides the problem into two parts. First a *Quasi-Realization Theorem* is shown, algorithmically. This extracts a quasi-realization from a modal tableau proof—a simpler thing to do than getting a realization proper. Then we show, again algorithmically, how to convert a quasi-realization into a realization. This part is independent of tableaus, or cut-free proofs in general, and applies to justification logics in general, not just to LP.

After presenting our algorithms, we implement them. We have used Prolog, because prototyping in this language is fairly simple. The reader is invited to reimplement in some more common language. A thorough description of the program is provided, which may help.

1.2 The Logic LP

This section contains a brief formulation of LP axiomatically. A semantics will not be needed in this paper. The language of LP is built from the following basic machinery, which comes from [1].

1. propositional variables, P, Q, ...
2. propositional constant, $\bot$
3. logical connective, $\supset$
4. justification variables, v_1, v_2, ...
5. justification constants, c_1, c_2, ...
6. function symbols ! (monadic), $\cdot$, + (binary)
7. operator symbol of the type $\langle term \rangle{:}\langle formula \rangle$

Justification terms are built up from justification variables and justification constants, using the function symbols. *Ground* justification terms are those without variables. *Formulas* are built up from propositional variables and the propositional constant $\bot$ using $\supset$ (with other connectives defined in the usual way), and an extra rule of formation: if t is a justification term and X is a formula then $t{:}X$ is a formula.

The formula $t{:}X$ can be read: "t is a justification of X." Justification constants intuitively represent proofs of basic, assumed truths. Justification variables in a formula can be thought of as implicitly universally quantified over proofs. If t is a justification of $X \supset Y$ and u is a justification of X, we should think of $t \cdot u$, the application of t to u, as a justification of Y. The operation ! is a checker: if t is a justification of X then $!t$ is a verification that t is such a justification. The operation + combines justifications in the sense that $t + u$ justifies all the things that t justifies plus all the things that u justifies.

The following axiom system for LP is from [1]. Axioms are specified by giving axiom schemas, and these are:

A0.	**Classical**	Enough classical propositional axiom schemes
A1.	**Application**	$t{:}(X \supset Y) \supset (s{:}X \supset (t \cdot s){:}Y)$
A2.	**Factivity**	$t{:}X \supset X$
A3.	**Justification Checker**	$t{:}X \supset\ !t{:}(t{:}X)$
A4.	**Weakening**	$s{:}X \supset (s+t){:}X$
		$t{:}X \supset (s+t){:}X$

Rules of inference are modus ponens, and a version of the necessitation rule, for axioms only.

R1.	**Modus Ponens**	$\vdash Y$ provided $\vdash X$ and $\vdash X \supset Y$
R2.	**Axiom Necessitation**	$\vdash c{:}X$ where X is an axiom $A0$ – $A4$ and c is a justification constant.

As usual, a proof is a finite sequence of formulas each of which is an axiom or comes from earlier terms by one of the rules of inference. A notion of *derivation* can be introduced either directly, or indirectly by defining $\Gamma \vdash X$ to mean that $(G_1 \wedge \ldots \wedge G_n) \supset X$ is a theorem for some finite subset $\{G_1, \ldots, G_n\}$ of Γ.

The specification of which constants are associated with which axioms for rule *R2* applications is called a *constant specification*. More formally, a constant specification is a set $\mathcal{C}$ whose members are of the form $c{:}A$ where c is a justification constant and A is an axiom. A proof uses constant specification $\mathcal{C}$ if each instance of Axiom Necessitation is in $\mathcal{C}$. A constant specification can be given ahead of time, or it could be created during the course of a proof. We will assume all constant specifications are *axiomatically appropriate*, every axiom is assigned at least one constant. In addition, all such assignments will be *injective*, no justification constant is used for more than one axiom. Many other conditions have been investigated, but we are not interested in constant specification details here.

The Artemov Realization Theorem is the subject of this report. If Z is any theorem of LP, and we replace every proof polynomial by $\Box$ (the *forgetful* projection), the result is a theorem of S4. This is easy to see: it is the case for each axiom of LP, and is preserved by the LP rules of derivation. The Realization Theorem, [1], is a converse to this.

Theorem 1.2.1 (Realization Theorem) *If Z is a theorem of S4, there is some way of replacing $\Box$ symbols with justification terms to produce a theorem of LP (provable using an injective constant specification). Moreover this can be done so that negative occurrences of $\Box$ in Z are replaced with distinct justification variables, and positive occurrences by justification terms that may involve those variables.*

Negative occurrences of justification variables can be thought of as inputs, and the justification terms involving them as outputs. Thus theorems of S4, in a sense, carry implicit constructive functional content which their embeddings into LP make explicit.

A fundamental result is the Lifting Lemma, from [1, 2], which says that proofs and derivations in LP can be internalized.

Theorem 1.2.2 (Lifting Lemma) *Suppose*

$$s_1{:}X_1, \ldots, s_n{:}X_n, Y_1, \ldots, Y_k \vdash Z$$

then there is a justification term $t(s_1, \ldots, s_n, y_1, \ldots, y_k)$ (where the y_i are justification variables) such that

$$s_1{:}X_1, \ldots, s_n{:}X_n, y_1{:}Y_1, \ldots, y_k{:}Y_k \vdash t(s_1, \ldots, s_n, y_1, \ldots, y_k){:}Z.$$

Corollary 1.2.3 *If Z has an LP proof, then for some ground proof polynomial t, $t{:}Z$ will have an LP proof.*

The proof polynomial t in the Corollary above can always be taken so that it does not involve the operator $+$. The standard proof, by induction on axiomatic derivation length, constructively produces such a polynomial.

Chapter 2

Tableaus

2.1 Classical Logic As Background

Tableaus are refutation proof systems. Informally, one assumes a formula X is not valid—false under some circumstance—and derives a formal contradiction. This establishes validity. For those logics having tableau systems, machinery varies quite a bit. At the heart, generally, is classical logic, and we sketch the ideas here as a (very) representative example.

For the time being, let us assume formulas are built up from propositional letters using just $\supset$ and $\neg$. Tableaus can use *signed* or *unsigned* formulas. Both have their particular advantages. Here it is most convenient to use signed formulas. For this purpose we introduce two special symbols, T and F, and specify that $T\,X$ and $F\,X$ are signed formulas if X is a formula. Of course the intended reading is that X is true, or false, but this might mean classically, or constructively, or at some possible world, depending on the logic in question. Since this is classical logic now, conventional truth and falsehood are the motivating intuitions behind T and F.

A tableau proof is a labeled binary tree, where the labels are signed formulas meeting special conditions. A proof of X begins with a tree having only a root node, labeled $F\,X$. Then a tree is 'grown' using *branch extension* rules, one for each propositional connective and each sign. The initial tree and each subsequent tree is a tableau. Here are the very simple branch extension rules for negation.

Negation

$$\frac{T\,\neg X}{F\,X} \qquad\qquad \frac{F\,\neg X}{T\,X}$$

Tableau rules can also introduce branching. Here are the two rules for implication, one of which is a branching rule.

Implication

$$\frac{T\,X \supset Y}{F\,X \mid T\,Y} \qquad\qquad \frac{F\,X \supset Y}{\begin{array}{c} T\,X \\ F\,Y \end{array}}$$

Tableaus are displayed as downward branching trees. Think of a tree as representing the disjunction of its branches, and a branch as representing the conjunction of the signed formulas on it. Since a node may be common to several branches, a formula labeling it, in effect, occurs as a constituent of several conjunctions, while being written only once. This amounts to a kind of structure sharing.

Important Note 2.1.1 Some variation is possible. The members of a tableau branch can be thought of as constituting a *set*, or a *multi-set*, or even a *sequence*. All have their uses. *Here we will treat branches as sets.*

A tableau expansion is often discussed temporally—one talks about the *stages* of constructing a tableau, meaning the stages of growing a tree. The rules given above are thought of as branch-lengthening rules. Thus, a branch containing $T\,\neg X$ can be lengthened by adding a new node to its end, with $F\,X$ as label. Likewise a branch containing $F\,X \supset Y$ can be lengthened with two new nodes, labeled $T\,X$ and $F\,Y$ (take the node with $F\,Y$ as the child of the one labeled $T\,X$). A branch containing $T\,X \supset Y$ can be split—its leaf is given a new left and a new right child, with one labeled $F\,X$, the other $T\,Y$.

Important Note 2.1.2 Tableau rules are *non-deterministic*. At each stage we choose a signed formula occurrence on a branch and apply a rule to it. Since the order of choice is arbitrary, there can be many tableaus for a single signed formula. Sometimes a prescribed order of rule application is imposed, but this is not basic to a tableau system.

Figure 2.1 shows the final stage of a tableau construction beginning with (that is, *for*) the signed formula $F\,(X \supset Y) \supset ((X \supset \neg Y) \supset \neg X)$. Numbers are shown for reference purposes. Items 2 and 3 are from 1 by $F\supset$; 4 and 5 are from 3 by $F\supset$; 6 is from 5 by $F\neg$; 7 and 8 are from 2 by $T\supset$; 9 and 10 are from 4 by $T\supset$; 11 is from 10 by $T\neg$.

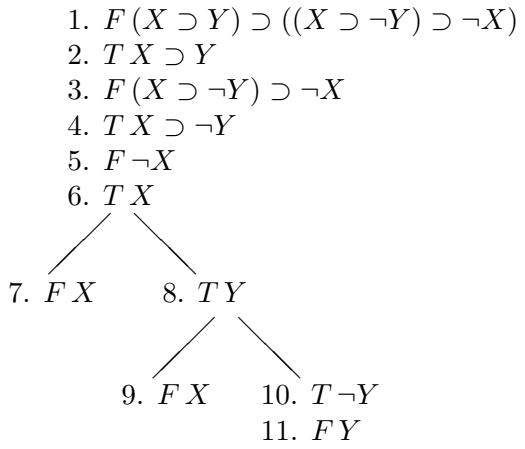

Figure 2.1: A Classical Tableau Example

Finally, the conditions for ending a proof. A branch is *closed* if it contains $T\,A$ and $F\,A$ for some formula A. A branch is also *closed* if it contains $T\,\bot$. If each branch is closed, the tableau is *closed*. A closed tableau for $F\,X$ is a tableau proof of X. The tableau displayed in Figure 2.1 is closed, so the formula $(X \supset Y) \supset ((X \supset \neg Y) \supset \neg X)$ has a tableau proof.

Important Note 2.1.3 For many logics, closure can be taken to be *atomic*. That is, a branch is closed if it contains $T\,P$ and $F\,P$ where P is atomic. This is so for all the logics we will consider here, though it is not true for all logics with tableau systems. We will require atomic closure.

Branch extension rules for classical connectives can be restricted to *single use*. That is, a tableau rule need never be applied to a signed formula occurrence on a branch more than once. The tableau

rules allow for multiple usage, but it is never necessary. (This is not true for all logics, however.) The tableau example shown above, in fact, follows a single use convention.

Important Note 2.1.4 All classical tableau rule applications will follow the single use convention.

2.2 Uniform Notation

Suppose other binary connectives in addition to $\supset$ are also allowed. Binary connectives have tableau rules with similar cases, so Smullyan introduced *uniform notation* to group cases together, [17, 18]. He did this for its theoretical advantages—proofs about tableau systems can often make use of uniform notation to reduce their case complexity. As it turns out, uniform notation also helps with tableau implementations, and we will make use of it. This has become common in the literature, and one often sees references to α or β cases. The notation extends to quantifiers too, but we will not need it here.

All signed propositional formulas of the forms $T\,X \circ Y$ and $F\,X \circ Y$, where $\circ$ is a binary connective, are grouped into two categories, those that act *conjunctively*, which are called α-*formulas*, and those that act *disjunctively*, which called β-*formulas*. For each α-formula two *components* are defined, which are denoted α_1 and α_2. Similarly, components β_1 and β_2 are defined for each β-formula. This is done in Table 2.1. The final two cases are a bit odd, but work just fine anyway.

Conjunctive			Disjunctive		
α	α_1	α_2	β	β_1	β_2
$T\,X \wedge Y$	$T\,X$	$T\,Y$	$F\,X \wedge Y$	$F\,X$	$F\,Y$
$F\,X \vee Y$	$F\,X$	$F\,Y$	$T\,X \vee Y$	$T\,X$	$T\,Y$
$F\,X \supset Y$	$T\,X$	$F\,Y$	$T\,X \supset Y$	$F\,X$	$T\,Y$
$F\,X \subset Y$	$F\,X$	$T\,Y$	$T\,X \subset Y$	$T\,X$	$F\,Y$
$F\,X \uparrow Y$	$T\,X$	$T\,Y$	$T\,X \uparrow Y$	$F\,X$	$F\,Y$
$T\,X \downarrow Y$	$F\,X$	$F\,Y$	$F\,X \downarrow Y$	$T\,X$	$T\,Y$
$T\,X \not\supset Y$	$T\,X$	$F\,Y$	$F\,X \not\supset Y$	$F\,X$	$T\,Y$
$T\,X \not\subset Y$	$F\,X$	$T\,Y$	$F\,X \not\subset Y$	$T\,X$	$F\,Y$
$T\,X \equiv Y$	$T\,X \supset Y$	$T\,Y \supset X$	$F\,X \equiv Y$	$F\,X \supset Y$	$F\,Y \supset X$
$F\,X \not\equiv Y$	$T\,X \supset Y$	$T\,Y \supset X$	$T\,X \not\equiv Y$	$F\,X \supset Y$	$F\,Y \supset X$

Table 2.1: α- and β-Formulas and Components

Now tableau rules can be given uniformly, for all binary connectives. This is done in Table 2.2.

$\dfrac{T\,\neg X}{F\,X}$	$\dfrac{F\,\neg X}{T\,X}$	$\dfrac{\alpha}{\begin{array}{l}\alpha_1\\ \alpha_2\end{array}}$	$\dfrac{\beta}{\beta_1 \mid \beta_2}$

Table 2.2: Classical Tableau Expansion Rules

Here's how to use the rules. Suppose we have $F\,P \wedge Q$ on a tableau branch. This is a β case, with $\beta_1 = F\,P$ and $\beta_2 = F\,Q$. The Tableau Expansion Rules say the branch can be split, with $F\,P$ on one fork and $F\,Q$ on the other. Note that the earlier rules for implication are subsumed in the uniform presentation above.

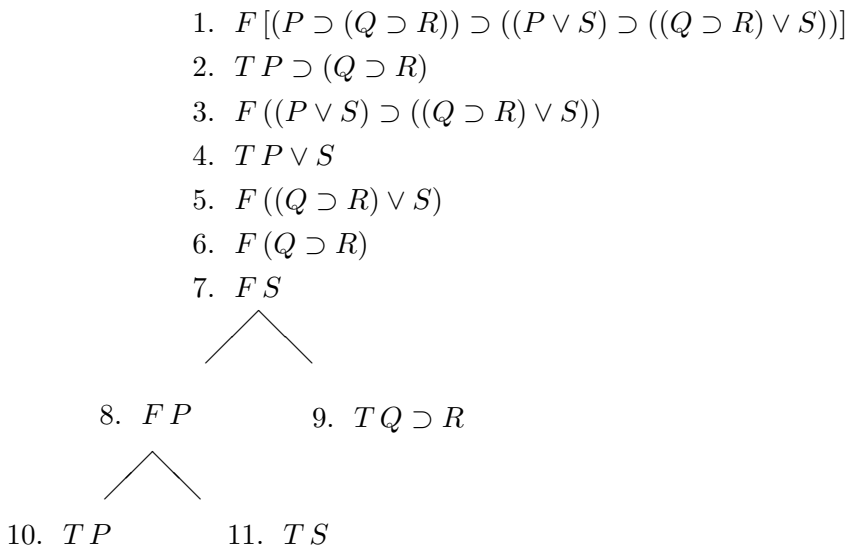

Figure 2.2: Classical Proof of $(P \supset (Q \supset R)) \supset ((P \vee S) \supset ((Q \supset R) \vee S))$

Figure 2.2 shows a tableau proof of $[(P \supset (Q \supset R)) \supset ((P \vee S) \supset ((Q \supset R) \vee S))]$. In it, 1 is F of the formula to be proved; 2 and 3 are from 1 by α; 4 and 5 are from 3 by α; 6 and 7 are from 5 by α; 8 and 9 are from 2 by β. 10 and 11 are from 4 by β. Reading from left to right, the branches are closed because of 8 and 10, 7 and 11, and 6 and 9. Notice that on one of the branches closure is on a non-atomic formula. This branch can be continued to yield atomic closure—we leave it to you.

Important Note 2.2.1 Tableaus and tableau implementations work fine with the whole list of binary connectives in Table 2.1, and our quasi-realization algorithm can handle the list except for $\equiv$ and $\not\equiv$. But most of the connectives are rare in practice so in our examples from now on, and in our implementation, we will assume the only binary connectives are $\wedge$, $\vee$, and $\supset$. We gave a full list primarily for reference and completeness of coverage.

2.3 Modal Tableaus

Uniform notation exists for modal operators as well: ν for necessity, π for possibility. But justification logics have made little use of possibility, so there is no particular advantage to modal uniform notation for us, and we will not use it. Syntactically $\Box$, but not $\Diamond$, is added to the language in the usual way.

There are several varieties of modal tableaus. We will use what are called *destructive* tableaus. The name comes from the fact that certain rules cause branch information to disappear. Such tableaus exist for K, T, D, D4, K4, S4, among other familiar logics, but not for S5. We will sketch rules for these systems.

2.4 Rules for K

Suppose S is a set of signed (modal) formulas. Let $S^\sharp = \{T\,X \mid T\,\Box X \in S\}$. There is only one modal rule for K, in addition to the classical propositional rules given earlier.

K Rule
$$\frac{S, F\,\Box X}{S^\sharp, F\,X}$$

This rule is applicable to a branch containing $F\,\Box X$, with S being the set of other formulas on the branch. *The entire branch is replaced with a new branch consisting of the members of $S^\sharp$, and $F\,X$.* Note that information is lost passing from S to $S^\sharp$, hence the name *destructive*.

Figure 2.3 shows a tableau proof, using the K-rules, of $(\Box X \wedge \Box Y) \supset \Box(X \wedge Y)$. It begins with several α rule applications. 1 is, of course, how the proof must start. 2 and 3 are from 1 by rule α. Likewise 4 and 5 are from 2 by rule α. Now $F\,\Box(X \wedge Y)$ occurs, so by the rules for K we may add $F\,X \wedge Y$, but replace the remaining set S of signed formulas with $S^\sharp$. This modifies the branch to consist of signed formulas 6, 7, and 8. Here 1, 2 and 3 have been deleted, 4 and 5 replaced by 6 and 7, then 8 added to replace 3. Now 8 is a β-formula and one more step yields a closed tableau.

1. $F\,(\Box X \wedge \Box Y) \supset \Box(X \wedge Y)$
2. $T\,\Box X \wedge \Box Y$
3. $F\,\Box(X \wedge Y)$
4. $T\,\Box X$
5. $T\,\Box Y$

---

6. $T\,X$
7. $T\,Y$
8. $F\,X \wedge Y$

Figure 2.3: K Proof of $(\Box X \wedge \Box Y) \supset \Box(X \wedge Y)$

Important Note 2.4.1 With classical propositional tableaus, *any* order of rule application must produce a proof if one exists, as long as eventually every applicable rule has been applied. This is not the case for K. If both $F\,\Box X$ and $F\,\Box Y$ are present, applying a rule to one eliminates the other, and it may be that only one of the two possibilities will lead to a proof. Now *backtracking* becomes critical to proof search.

2.5 Some Other Modal Logics

Several other standard modal logics have destructive tableau systems. Sometimes the definition of $S^\sharp$ needs some modification. Variations on modal rules are introduced. The $S^\sharp$ operations for several logics are given in Table 2.3 and the modal rules are given in Table 2.4.

Figure 2.4 shows a proof, using the S4 rules, of $\Box X \supset \Box(\Box X \vee Y)$. Lines 2 and 3 are from 1 by α. Next an S4 modal rule is applied to $F\,\Box(\Box X \vee Y)$, adding 4 while replacing S by $S^\sharp$ eliminates 1 and 3. Now an α-rule application to 4 adds 5 and 6, and produces a closed tableau, though not an atomically closed one. Continuing, we apply an S4 modal rule again, to 5, adding 7 while eliminating 4, 5, and 6. Finally, applying the second S4 modal rule to 2 adds 8, and we have atomic closure.

Logic	Definition of $S^\sharp$
K	$\{T\,X \mid T\,\Box X \in S\}$
K4	$\{T\,X, T\,\Box X \mid T\,\Box X \in S\}$
T	$\{T\,X \mid T\,\Box X \in S\}$
S4	$\{T\,\Box X \mid T\,\Box X \in S\}$
D	$\{T\,X \mid T\,\Box X \in S\}$
D4	$\{T\,X, T\,\Box X \mid T\,\Box X \in S\}$

Table 2.3: Definitions for $S^\sharp$

Logic	Rule
K, K4, T, S4, D, D4	$\dfrac{S, F\,\Box X}{S^\sharp, F\,X}$
K, K4	no additional rules
T, S4	$\dfrac{T\,\Box X}{T\,X}$
D, D4	$\dfrac{S}{S^\sharp}$

Table 2.4: Modal Rules

Important Note 2.5.1 The rule $S, F\,\Box X \Rightarrow S^\sharp, F\,X$ is single usage by default. Applying it with $F\,\Box X$ eliminates the formula, so it cannot be applied a second time. The rule $T\,\Box X \Rightarrow T\,X$ for T is also single usage, but for S4 things are trickier. For S4, if $T\,\Box X \Rightarrow T\,X$ is applied to a signed formula occurrence it need not be applied again, *until the rule* $S, F\,\Box X \Rightarrow S^\sharp, F\,X$ *has been applied.* The intuition is simple: the destructive rule might eliminate the consequent of $T\,\Box X \Rightarrow T\,X$ but for S4 it will not eliminate the premise, so a new application may be useful.

2.6 Annotated Formulas and Tableaus

We will be mapping modal formulas to formulas of justification logic. This requires that we keep track of the various *occurrences* of $\Box$. In [6] we introduced *annotated formulas* to address this issue; we use a simpler version now. An *annotated modal formula* is like a standard modal formula except for the following.

1. Instead of a single modal operator $\Box$ there is an infinite family, $\Box_1$, $\Box_2$, ..., called *indexed* modal operators. Formulas are built up as usual, but using indexed modal operators instead of $\Box$. We assume that in an annotated formula, *no index occurs twice.*

2. If A is an annotated formula, and A' is the result of replacing all indexed modal operators, $\Box_n$, with $\Box$, regardless of index, then A' is a conventional modal formula. We say A is an *annotated version* of A', and A' is an *unannotated version* of A.

Annotations are purely for bookkeeping purposes. Here are details. The α/β classification is exactly as with unannotated formulas, as is the definition of components. Thus, for instance, $T\,\Box_1 P \wedge \Box_2 Q$ counts as an α, with $\alpha_1 = T\,\Box_1 P$ and $\alpha_2 = T\,\Box_2 Q$. In tableau constructions, branch extension rules apply to annotated formulas exactly as to unannotated ones. The definition of $S^\sharp$

1. $F\,\Box X \supset \Box(\Box X \vee Y)$
2. $T\,\Box X$
3. $F\,\Box(\Box X \vee Y)$

---

2. $T\,\Box X$
4. $F\,\Box X \vee Y$
5. $F\,\Box X$
6. $F\,Y$

---

2. $T\,\Box X$
7. $F\,X$
8. $T\,X$

Figure 2.4: S4 proof of $\Box X \supset \Box(\Box X \vee Y)$

for S4 becomes the following. $S^\sharp = \{T\,\Box_i X \mid T\,\Box_i X \in S\}$, and similarly for other modal logics. And since we are requiring atomic closure, closure conditions are not affected by annotations.

Given all this, Figure 2.5 is an annotated version of the S4 proof shown in Figure 2.4. Every S4 tableau proof can be turned into an annotated proof, simply by annotating the modal operators appearing in the root, and then propagating these annotations downward through the tree.

1. $F\,\Box_1 X \supset \Box_2(\Box_3 X \vee Y)$
2. $T\,\Box_1 X$
3. $F\,\Box_2(\Box_3 X \vee Y)$

---

2. $T\,\Box_1 X$
4. $F\,\Box_3 X \vee Y$
5. $F\,\Box_3 X$
6. $F\,Y$

---

2. $T\,\Box_1 X$
7. $F\,X$
8. $T\,X$

Figure 2.5: S4 proof of $\Box_1 X \supset \Box_2(\Box_3 X \vee Y)$

2.7 Changing the Tableau Representation

So far tableaus have been trees, and formula occurrences could be common to multiple branches. While this has advantages for some purposes, it does not when our quasi-realization algorithm is introduced. We will be associating a set of quasi-realizers with each signed formula occurrence in a tableau. How that is done depends on the history of the branch containing a given occurrence. If an occurrence is common to more than one branch it has more than one history, and things become ambiguous. Our simple solution is to change the way tableaus are represented, something that also brings us much closer to the data structure used in our Prolog implementation.

From now on a tableau is not thought of as a tree, but rather as the set of its branches. Each branch, in turn, is the set of signed formulas on it. When we write a branch as $\mathcal{B}, Z$, or more graphically $\overset{\mathcal{B}}{Z}$, we mean it is the set whose members are those of $\mathcal{B}$, together with signed formula Z. *This notation assumes that Z is not part of $\mathcal{B}$.* We now reformulate the S4 tableau rules in this

style, building in the notion of single-usage for tableau rules. Of course rules for other modal logics could also be presented, but S4 will be sufficiently representative. Here are the formal details. They apply equally well to tableaus of signed formulas or of annotated signed formulas.

Definition 2.7.1 (Tableau Revised) A tableau is a finite set of finite sets (called branches) of signed formulas—see Important Note 2.1.1. A branch is closed if it contains $T\,P$ and $F\,P$ for some atomic P, or if it contains $T\,\bot$ (see Important Note 2.1.3). A tableau is closed if each of its branches is closed. We say a signed formula is on a branch if it is a member of it, and a branch is in a tableau if it is a member of it. A *tableau proof* of X is a sequence of tableaus, beginning with a single branch tableau where that branch contains only $F\,X$, continuing using the Branch Extension Rules given in Tables 2.5 and 2.6, and ending with a closed tableau.

$$\begin{array}{cccc}
\mathcal{B} & \mathcal{B} & \mathcal{B} & \mathcal{B} \\
T\,\neg X & F\,\neg X & \alpha & \beta \\
\hline
\mathcal{B} & \mathcal{B} & \mathcal{B} & \mathcal{B} \mid \mathcal{B} \\
F\,X & T\,X & \alpha_1 & \beta_1 \mid \beta_2 \\
 & & \alpha_2 &
\end{array}$$

Table 2.5: Classical Branch Extension Rules Revised

For example, the β rule in Table 2.5 is to be read as follows. If a tableau has $\mathcal{B}, \beta$ as a branch, then the result of removing the branch from the tableau and replacing it with two branches, $\mathcal{B}, \beta_1$ and $\mathcal{B}, \beta_2$ is another tableau, which we will call a *successor* of the original tableau. Similarly for the other rules. Note that the new branches do not contain β, but have β_1 and β_2 instead. In effect, we remove a signed formula once a classical rule has been applied to it. This is how we enforce single usage—see Important Note 2.1.4. That branches do not share any common parts is essential here.

There is one misleading aspect to the notation above. In the rule for $T\,\neg$ for instance, it may happen that $F\,X$ already occurs in $\mathcal{B}$, in which case the display of $\mathcal{B}, F\,X$ below the line is not correct—it should be simply $\mathcal{B}$. We allow this mild abuse, rather than complicating notation.

$$\begin{array}{cc}
\mathcal{B} & \mathcal{B} \\
T\,\Box X & F\,\Box X \\
\hline
\mathcal{B} & \mathcal{B}^\sharp \\
\cancel{T\,\Box X} & F\,X \\
T\,X &
\end{array}$$

$$\mathcal{B}^\sharp = \{T\,\Box X \mid T\,\Box X \in \mathcal{B} \text{ or } \cancel{T\,\Box X} \in \mathcal{B}\}$$

Table 2.6: S4 Modal Branch Extension Rules Revised

Single usage is a bit trickier for modal rules. As Important Note 2.5.1 points out, single usage for the $F\,\Box$ rule is automatic, but for the $T\,\Box$ rule of S4 single usage only applies until the next application of the $F\,\Box$ rule. We build this into Table 2.6 by crossing off an occurrence of $T\,\Box X$ when a rule has been applied to it, and providing no rule that has a crossed off signed formula as a trigger. A cross off mark is removed, as part of the definition of $\mathcal{B}^\sharp$, when an $F\,\Box$ rule is applied.

Figure 2.6 shows an example of a tableau proof of $\Box(P \supset Q) \supset (\Box P \supset \Box Q)$, where a tableau is a set of its branches, which is a set of its signed formulas. Each line is a tableau represented as a set of sets of signed formulas, and the passage from line to line is according to the Branch Extension Rules for S4. (It is not the shortest tableau proof for this formua.) The final tableau is closed.

1. $\{\{F\,\Box(P \supset Q) \supset (\Box P \supset \Box Q)\}\}$
2. $\{\{T\,\Box(P \supset Q), F\,\Box P \supset \Box Q\}\}$
3. $\{\{\underline{T\,\Box(P \supset Q)}, T\,P \supset Q, F\,\Box P \supset \Box Q\}\}$
4. $\{\{\underline{T\,\Box(P \supset Q)}, T\,P \supset Q, T\,\Box P, F\,\Box Q\}\}$
5. $\{\{T\,\Box(P \supset Q), T\,\Box P, F\,Q\}\}$
6. $\{\{\underline{T\,\Box(P \supset Q)}, T\,P \supset Q, T\,\Box P, F\,Q\}\}$
7. $\{\{\underline{T\,\Box(P \supset Q)}, F\,P, T\,\Box P, F\,Q\}, \{\underline{T\,\Box(P \supset Q)}, T\,Q, T\,\Box P, F\,Q\}\}$
8. $\{\{\underline{T\,\Box(P \supset Q)}, F\,P, \underline{T\,\Box P}, T\,P, F\,Q\}, \{\underline{T\,\Box(P \supset Q)}, T\,Q, T\,\Box P, F\,Q\}\}$

Figure 2.6: Tableau As Set Of Sets

2.8 Final Remarks

Semantic tableaus are closely related to, well, semantics. But this is not a treatise on tableaus, so I'm omitting all this. However, there is one proof-theoretic way of understanding tableaus that has relevance to the quasi-realization algorithm that will be presented in Chapter 3.

Definition 2.8.1 Let $\mathcal{B}$ be a branch of a tableau (classical, modal, ...). By the *associated formula* for $\mathcal{B}$ we mean $\bigwedge \mathcal{B}_T \supset \bigvee \mathcal{B}_F$ where $\mathcal{B}_T = \{X \mid T\,X \text{ is on } \mathcal{B}\}$ and $\mathcal{B}_F = \{X \mid F\,X \text{ is on } \mathcal{B}\}$.

Suppose we have a tableau system for a modal logic considered in this chapter, and we also have an axiomatic formulation. If a tableau branch is closed, or can be continued to closure, the formula associated with that branch will be provable in that axiom system. Here is how to show this. If a branch is closed it is obvious, since both $T\,P$ and $F\,P$ will be on the branch, which makes the associated formula a tautology. With a modest amount of work, one can show the following. If branch $\mathcal{B}$ extends in one step to a branch or branches having provable associated formulas, then $\mathcal{B}$ itself has a provable associated formula. Then a kind of 'backward induction' establishes the result for all closable branches.

If X has a tableau proof, a tableau for $F\,X$ closes, so the associated formula for the only branch of the initial tableau is axiomatically provable. But this this branch contains only $F\,X$, so the associated formula is $\bigwedge \emptyset \supset \bigvee X$, or $\top \supset X$, or equivalently, X, and hence X is axiomatically provable. This is a version of soundness for tableaus relative to axiomatics. The idea will turn up again.

Chapter 3

Quasi-Realizations

3.1 The Basic Idea

The algorithm for computing realizations, presented in this report, divides into two halves. The first half constructs an intermediate object, a *quasi-realization*, from a tableau proof. The second half converts a quasi-realization into a proper realization, and is discussed in Chapter 4. The construction of quasi-realizations is logic dependent, though differences between many basic modal logics often are rather minor. Input to this algorithm is the steps in a tableau proof. Similar algorithms can also be constructed that use Gentzen system proofs, or nested sequent proofs. The input to the quasi-realization to realization algorithm is a quasi-realization, not a formal proof, and the construction is independent of the particular logic involved.

To keep things concrete in the discussion of this chapter, we assume the modal logic considered is S4, and the justification logic is LP. Similar constructions apply to other modal/justification combinations too, and how to do this should be relatively obvious.

Informally the goal is to associate a quasi-realization with each signed formula occurrence in a tableau proof. This quasi-realization is constructed according to how the branch on which the signed formula appears is continued to closure. The problem is that a given signed formula occurrence can be on more than one branch, appearing before the branch splits. A quasi-realization computed on one branch might be different than a quasi-realization computed on another. If we were dealing with realizations, a merging solution to this problem would involve the + operation and substitution, and would be of some complexity since the entire nested structure of the formulas involved would need to be taken into consideration. Our approach bypasses this problem by allowing a *set* of quasi-realizations rather than insisting on a single one. In fact + does not appear until we reach the quasi-realization to realization algorithm.

3.2 Quasi-Realizations Defined

We define a map from *annotated* signed modal formulas to sets of signed LP formulas. From now on we assume that v_1, v_2, ... is an enumeration of all justification variables of LP with no variable repeated, fixed once and for all. Case 4 of the definition below always uses v_k in quasi-realizations where the sign is T and $\Box_k$ is involved. In case 2 two conjunctive, or α, signed formulas are mentioned. For one we use α with α_1 and α_2 as components. For the other we use α' with α'_1 and α'_2 as components. Similarly for disjunctive, or β, signed formulas.

Definition 3.2.1 The mapping $\langle\!\langle \cdot \rangle\!\rangle$ is defined recursively on the set of signed annotated modal formulas.

1. If A is atomic, $\langle\!\langle T\,A \rangle\!\rangle = \{T\,A\}$ and $\langle\!\langle F\,A \rangle\!\rangle = \{F\,A\}$.
2. $\langle\!\langle T\,\neg A \rangle\!\rangle = \{T\,\neg U \mid F\,U \in \langle\!\langle F\,A \rangle\!\rangle\}$.
   $\langle\!\langle F\,\neg A \rangle\!\rangle = \{F\,\neg U \mid T\,U \in \langle\!\langle T\,A \rangle\!\rangle\}$.
3. $\langle\!\langle \alpha \rangle\!\rangle = \{\alpha' \mid \alpha'_1 \in \langle\!\langle \alpha_1 \rangle\!\rangle \text{ and } \alpha'_2 \in \langle\!\langle \alpha_2 \rangle\!\rangle\}$.
   $\langle\!\langle \beta \rangle\!\rangle = \{\beta' \mid \beta'_1 \in \langle\!\langle \beta_1 \rangle\!\rangle \text{ and } \beta'_2 \in \langle\!\langle \beta_2 \rangle\!\rangle\}$.
4. $\langle\!\langle T\,\Box_n A \rangle\!\rangle = \{T\,v_n{:}U \mid T\,U \in \langle\!\langle T\,A \rangle\!\rangle\}$.
   $\langle\!\langle F\,\Box_n A \rangle\!\rangle = \{F\,t{:}(U_1 \vee \ldots \vee U_k) \mid F\,U_1, \ldots, F\,U_k \in \langle\!\langle F\,A \rangle\!\rangle \text{ and } t \text{ is any justification term}\}$.
5. The mapping is extended to *sets* of signed annotated formulas by letting $\langle\!\langle S \rangle\!\rangle = \cup\{\langle\!\langle Z \rangle\!\rangle \mid Z \in S\}$.

Members of $\langle\!\langle Z \rangle\!\rangle$ are called *quasi-realizers* of Z.

Example 3.2.2 Suppose t, u, and w are justification terms and P and Q are atomic formulas. Here are some quasi-realization calculations, leading up to $F\,\Box_1(\Box_2 P \vee \neg\Box_3 Q)$. We do not produce *all* quasi-realizations; the set in this case would be infinite. Most of the reasoning is obvious; we discuss only one case. $F\,\Box_2 P \vee \neg\Box_3 Q$, in item 5, is an α, with $\alpha_1 = F\,\Box_2 P$ and $\alpha_2 = F\,\neg\Box_3 Q$. By items 2 and 3, we can take $\alpha'_1 = F\,t{:}P$ and $\alpha'_2 = F\,\neg v_3{:}Q$, and then $\alpha' = F\,t{:}P \vee \neg v_3{:}Q$, which is taken to be one of the members of $\langle\!\langle F\,\Box_2 P \vee \neg\Box_3 Q \rangle\!\rangle$.

1. $\{F\,P\} = \langle\!\langle F\,P \rangle\!\rangle$ and $\{T\,Q\} = \langle\!\langle T\,Q \rangle\!\rangle$
2. $\{F\,t{:}P, F\,u{:}P\} \subseteq \langle\!\langle F\,\Box_2 P \rangle\!\rangle$
3. $\{T\,v_3{:}Q\} = \langle\!\langle T\,\Box_3 Q \rangle\!\rangle$
4. $\{F\,\neg v_3{:}Q\} = \langle\!\langle F\,\neg\Box_3 Q \rangle\!\rangle$
5. $\{F\,t{:}P \vee \neg v_3{:}Q, F\,u{:}P \vee \neg v_3{:}Q\} \subseteq \langle\!\langle F\,\Box_2 P \vee \neg\Box_3 Q \rangle\!\rangle$
6. $\{F\,t{:}((t{:}P \vee \neg v_3{:}Q) \vee (u{:}P \vee \neg v_3{:}Q)), F\,w{:}(u{:}(P \vee \neg v_3{:}Q)\} \subseteq \langle\!\langle F\,\Box_1(\Box_2 P \vee \neg\Box_3 Q) \rangle\!\rangle$

In Section 3.3 we give an algorithm which will establish the following.

Theorem 3.2.3 *Let X be an annotated modal formula. Given a tableau proof of X in* S4*, a finite set $\{F\,Q_1, \ldots, F\,Q_k\}$ of quasi-realizers for $F\,X$ can be constructed so that $Q_1 \vee \ldots \vee Q_k$ is a theorem of* LP*.*

Similar results can also be shown for the other modal logics for which tableau systems were given in Sections 2.4 and 2.5.

3.3 Mixed Tableaus

We now introduce what we call mixed tableaus, which unite modal features with justification logic features. They are based on tableaus as defined in Section 2.7, using a set of sets representation.

Definition 3.3.1 A *mixed S4 tableau* is like a tableau except that members of branches are not signed formulas, but are pairs (M, S) where M is a signed annotated modal formula and S is a finite set of signed justification formulas. There are two requirements that must be met.

1. If (M, S) occurs in a mixed tableau, it is required that $S \subseteq \langle\langle M \rangle\rangle$.
2. If, in a mixed tableau, we replace each entry (M, S) by just M, the result must be an annotated S4 tableau.

In a mixed tableau, we refer to M as the *modal part* of (M, S), and to S as the *justification part* of (M, S). We say a mixed tableau $\mathcal{T}^{mix}$ is an *expansion* of an S4 tableau $\mathcal{T}$ if $\mathcal{T}$ results from $\mathcal{T}^{mix}$, as in item 2 above, by eliminating the justification parts of node labels.

Informally, a mixed tableau expands an S4 tableau by associating a set of quasi-realizers to each signed annotated modal formula appearing in it. Before reading the next definition, it would be helpful to reread the Final Remarks in Section 2.8.

Definition 3.3.2 Let $\mathcal{B}$ be a branch of a mixed tableau. By the *associated justification formula* for $\mathcal{B}$ we mean $\bigwedge \mathcal{B}_T^{just} \supset \bigvee \mathcal{B}_F^{just}$ where $\mathcal{B}_T^{just}$ is the set of all justification formulas X such that $T\,X$ occurs in the justification part of some member of $\mathcal{B}$ and $\mathcal{B}_F^{just}$ is the set of X such that $F\,X$ occurs in the justification part of some member of $\mathcal{B}$.

We say a mixed S4 tableau branch is *justification sound* provided that its associated justification formula is provable in axiomatic LP. We say a mixed S4 tableau is justification sound if each branch is.

The heart of our quasi-realization work is the following.

Theorem 3.3.3 *Let $\mathcal{T}$ be an annotated S4 tableau that can be continued to one that is closed (or is closed already). Then $\mathcal{T}$ has a mixed tableau expansion $\mathcal{T}^{mix}$ that is justification sound, where $\mathcal{T}^{mix}$ can be algorithmically constructed from any closed modal tableau extending $\mathcal{T}$.*

This Theorem immediately gives us Theorem 3.2.3, which we re-state here for convenience.

Repeat of Theorem 3.2.3 Let X be an annotated modal formula. Given a tableau proof of X in S4, that is, given a closed S4 tableau starting with $F\,X$, a finite set $\{F\,Q_1, \ldots, F\,Q_k\}$ of quasi-realizers for $F\,X$ can be constructed so that $Q_1 \vee \ldots \vee Q_k$ is a theorem of LP.

Here is the argument. Suppose X is an annotated modal formula, and we have a closed S4 tableau proof for X. The construction of that proof begins with the single-branch modal tableau consisting of just a root node, labeled $F\,X$. Since this trivial tableau can be continued to a closed tableau, by Theorem 3.3.3 it can be expanded to a mixed tableau that is justification sound. Such a mixed tableau must consist of just a root node, labeled $(F\,X, \{F\,Q_1, \ldots, F\,Q_k\})$, where $\{F\,Q_1, \ldots, F\,Q_k\} \subseteq \langle\langle F\,X \rangle\rangle$. Since this expanded tableau is justification sound, the formula $\bigwedge \emptyset \supset \bigvee \{Q_1, \ldots, Q_k\}$ is axiomatically LP provable. That is, $Q_1 \vee \ldots \vee Q_k$ is provable, where $Q_1, \ldots, Q_k$ are quasi-realizers for $F\,X$.

3.4 The Quasi-Realization Algorithm

This section contains an algorithm for constructing justification sound mixed tableaus from closed S4 tableaus, followed by an example. In Section 3.5 a proof of the correctness of the algorithm

is given, and this establishes Theorem 3.3.3. The construction is a kind of 'backward induction'. Specifically, the strategy followed is this. Suppose $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$ is a sequence of annotated S4 tableaus, in which each arises from the preceding by a single application of an S4 branch extension rule, as given in Section 2.7. Suppose also that $\mathcal{T}_k$ is closed. We show $\mathcal{T}_k$ has a mixed tableau expansion that is justification sound. Then, using this, we show the same for $\mathcal{T}_{k-1}$, then for $\mathcal{T}_{k-2}$, and so on back to $\mathcal{T}_1$. A bit more properly, the algorithm produces a mixed tableau expansion for each $\mathcal{T}_i$; the correctness proof in the following section shows that it must be justification sound.

A branch extension rule application modifies only one branch—all others remain unchanged. Consequently the algorithm is stated in terms of branch extension rules applied to single branches. It is understood that the rest of the mixed tableau being constructed does not change, so unaffected branches are not explicitly displayed.

Tableaus are understood as sets of branches, with branches being sets of signed annotated formulas, as in Section 2.7. We make use of the notion convention introduced there, where $\mathcal{B}, Z$, $\mathcal{B}$ or Z, is a branch consisting of the members of $\mathcal{B}$, and Z (which is understood not to occur in $\mathcal{B}$).

The idea is to *expand* branches of an annotated S4 tableau so they become branches of a mixed tableau. Generally if $\mathcal{B}$ is an S4 tableau branch, we will write $\mathcal{B}^E$ to denote an expansion of it to a mixed tableau branch. Thus each signed annotated formula M in $\mathcal{B}$ is transformed into a pair (M, S) in $\mathcal{B}^E$ so that $S \subseteq \langle\!\langle M \rangle\!\rangle$. Of course $\mathcal{B}^E$ is not unique—it is simply some expansion. In one case of the algorithm more than one branch expansion must be referenced, and we use $\mathcal{B}^{E_1}$ and $\mathcal{B}^{E_2}$ as notation. We write $\mathcal{B} \xrightarrow{exp} \mathcal{B}^E$ to indicate that annotated S4 tableau branch $\mathcal{B}$ expands to mixed tableau branch $\mathcal{B}^E$.

If $\mathcal{B}^{E_1}$ and $\mathcal{B}^{E_2}$ are both expansions of the same branch, $\mathcal{B}$, by $\mathcal{B}^{E_1} \dot{\cup} \mathcal{B}^{E_2}$ we mean the mixed tableau branch consisting of all $(M, S_1 \cup S_2)$ where $(M, S_1) \in \mathcal{B}^{E_1}$ and $(M, S_2) \in \mathcal{B}^{E_2}$.

In a few of the algorithm cases we refer to a *trivial expansion*. A trivial expansion of a signed formula M is (M, S) where S is *any* finite set such that $S \subseteq \langle\!\langle M \rangle\!\rangle$. A trivial expansion of an S4 branch replaces each member with a trivial expansion. When we come to our implementation, a particular easily computed trivial expansion will be used, but the details don't matter for now.

The algorithm is stated schematically. We give a reading of the α Case as a representative example of how the algorithm notation should be understood. The idea is, we say how to expand the S4 tableau branch $\mathcal{B}, \alpha$ provided we already know how to expand $\mathcal{B}, \alpha_1, \alpha_2$. So, assume we have an expansion for S4 tableau branch $\mathcal{B}, \alpha_1, \alpha_2$, where $\mathcal{B}$ expands to $\mathcal{B}^E$, α_1 expands to (α_1, S_1), and α_2 expands to (α_2, S_2). Then S4 tableau branch $\mathcal{B}, \alpha$ expands to $\mathcal{B}^E, (\alpha, S)$, where S consists of all α signed formulas for which $\alpha_1 \in S_1$ and $\alpha_2 \in S_2$. (In the schematic we used α', α'_1, and α'_2 in characterizing S, simply because α, α_1, and α_2 were already in use to designate members of S4 tableau branches.)

Recall that v_1, v_2, ... is a fixed enumeration of all justification variables of LP with no variable repeated.

Algorithm 3.4.1 (Quasi-Realization Construction)

Atomic Cases

$$\begin{array}{c} \mathcal{B} \\ T\,P \\ F\,P \end{array} \xrightarrow{exp} \begin{array}{c} \mathcal{B}^E \\ (T\,P, \{T\,P\}) \\ (F\,P, \{F\,P\}) \end{array} \qquad \text{where } \mathcal{B}^E \text{ trivially expands } \mathcal{B}$$

$$\begin{array}{c} \mathcal{B} \\ T\,\bot \end{array} \xrightarrow{exp} \begin{array}{c} \mathcal{B}^E \\ (T\,\bot, \{T\,\bot\}) \end{array} \qquad \text{where } \mathcal{B}^E \text{ trivially expands } \mathcal{B}$$

α Cases

$$\dfrac{\begin{array}{ccc}\mathcal{B} & \xrightarrow{exp} & \mathcal{B}^E \\ \alpha_1 & & (\alpha_1, S_1) \\ \alpha_2 & & (\alpha_2, S_2)\end{array}}{\begin{array}{ccc}\mathcal{B} & \xrightarrow{exp} & \mathcal{B}^E \\ \alpha & & (\alpha, S)\end{array}} \quad \text{where } S = \{\alpha' \mid \alpha'_1 \in S_1 \text{ and } \alpha'_2 \in S_2\}$$

β Cases

$$\dfrac{\begin{array}{ccc}\mathcal{B} & \xrightarrow{exp} & \mathcal{B}^{E_1} \\ \beta_1 & & (\beta_1, S_1)\end{array} \qquad \begin{array}{ccc}\mathcal{B} & \xrightarrow{exp} & \mathcal{B}^{E_2} \\ \beta_2 & & (\beta_2, S_2)\end{array}}{\begin{array}{ccc}\mathcal{B} & \xrightarrow{exp} & \mathcal{B}^E \\ \beta & & (\beta, S)\end{array}} \quad \begin{array}{l}\text{where } S = \{\beta' \mid \beta'_1 \in S_1 \text{ and } \beta'_2 \in S_2\} \\ \text{and } \mathcal{B}^E = \mathcal{B}^{E_1} \dot{\cup} \mathcal{B}^{E_2}\end{array}$$

Negation Cases

$$\dfrac{\begin{array}{ccc}\mathcal{B} & \xrightarrow{exp} & \mathcal{B}^E \\ F\,X & & (F\,X, S_0)\end{array}}{\begin{array}{ccc}\mathcal{B} & \xrightarrow{exp} & \mathcal{B}^E \\ T\,\neg X & & (T\,\neg X, S)\end{array}} \quad \text{where } S = \{T\,\neg Z \mid F\,Z \in S_0\}$$

$$\dfrac{\begin{array}{ccc}\mathcal{B} & \xrightarrow{exp} & \mathcal{B}^E \\ T\,X & & (T\,X, S_0)\end{array}}{\begin{array}{ccc}\mathcal{B} & \xrightarrow{exp} & \mathcal{B}^E \\ F\,\neg X & & (F\,\neg X, S)\end{array}} \quad \text{where } S = \{F\,\neg Z \mid T\,Z \in S_0\}$$

$T\,\Box$ Case

$$\dfrac{\begin{array}{ccc}\mathcal{B} & \xrightarrow{exp} & \mathcal{B}^E \\ \cancel{T\,\Box_k X} & & (\cancel{T\,\Box_k X}, S_0) \\ T\,X & & (T\,X, S_1)\end{array}}{\begin{array}{ccc}\mathcal{B} & \xrightarrow{exp} & \mathcal{B}^E \\ T\,\Box_k X & & (T\,\Box_k X, S)\end{array}} \quad \text{where } S = S_0 \cup \{T\,v_k{:}Z \mid T\,Z \in S_1\}$$

$F\,\Box$ Case

$$\dfrac{\begin{array}{ccc}\mathcal{B}^\sharp & \xrightarrow{exp} & (\mathcal{B}^\sharp)^E \\ F\,X & & (F\,X, S_0)\end{array}}{\begin{array}{ccc}\mathcal{B}^\sharp & & (\mathcal{B}^\sharp)^E \\ \mathcal{B} - \mathcal{B}^\sharp & \xrightarrow{exp} & (\mathcal{B} - \mathcal{B}^\sharp)^E \\ F\,\Box_n X & & (F\,\Box_n X, \{F\,t{:}\bigvee S\})\end{array}} \quad \begin{array}{l}\text{where } (\mathcal{B} - \mathcal{B}^\sharp)^E \text{ trivially expands } \mathcal{B} - \mathcal{B}^\sharp, \\ S = \{Z \mid F\,Z \in S_0\}, \\ \bigwedge \mathcal{A} \supset \bigvee S \text{ is the associated justification} \\ \text{formula for branch } ((\mathcal{B}^\sharp)^E, (F\,X, S_0) \\ \text{and } \vdash_{\mathsf{LP}} \bigwedge \mathcal{A} \supset t{:}\bigvee S\end{array}$$

In the $F\,\Box$ case, $\bigwedge \mathcal{A} \supset \bigvee S$ appears as the associated justification formula for the branch $((\mathcal{B}^\sharp)^E, (F\,X, S_0)$. In fact, $\mathcal{A} = ((\mathcal{B}^\sharp)^E)^{just}_T$, using the notation of Definition 3.3.2, but such detailed notation distracts from the basic idea and we have suppressed it here. The existence of a term t such that $\vdash_{\mathsf{LP}} \bigwedge \mathcal{A} \supset t{:}\bigvee S$ will be guaranteed by the Lifting Lemma. Also note that the combination $\mathcal{B}^\sharp$ and $\mathcal{B} - \mathcal{B}^\sharp$ below the line simply amounts to $\mathcal{B}$, though the separation is useful for our purposes.

For each α signed formula, α_1 and α_2 are completely specified, but the converse direction is ambiguous if all binary connectives are allowed. For instance, if $\alpha_1 = T\,X$ and $\alpha_2 = T\,Y$, α could be either $T\,X \wedge Y$ or $F\,X \uparrow Y$, according to Table 2.1. Similarly for β. However, we have restricted

our attention to $\wedge$, $\vee$, and $\supset$ exclusively, and for these the values of α_1 and α_2 completely determine α. Again, similarly for β. This means the definitions of S in the α and β Cases of the Algorithm can have the simple forms that they do. If we allowed other connectives we would have to specify the 'type' of α or β as well.

We give an example to illustrate how Algorithm 3.4.1 works. Figure 3.1 shows an S4 proof of the annotated formula $(\Box_1 A \vee \Box_2 B) \supset \Box_3(A \vee B)$, using the representation of tableaus described in Section 2.7. Each numbered item should be thought of as the set of signed formulas making up a tableau branch. A detailed description follows. 1 is the initial single branch tableau. Single branch tableau 2 follows from 1 by α. A β rule application creates a tableau with two branches, 3 and 4. Modal rule applications on $F\,\Box_3(A \vee B)$ in 3 and 4 produce the two-branched tableau having branches 5 and 6. Modal rule applications on $T\,\Box_1 A$ and $T\,\Box_2 B$ in these give the two branches 7 and 8. Finally, α rule applications give the two branches 9 and 10, both of which are closed.

1. $F\,(\Box_1 A \vee \Box_2 B) \supset \Box_3(A \vee B)$

2. $T\,\Box_1 A \vee \Box_2 B$
   $F\,\Box_3(A \vee B)$

3. $T\,\Box_1 A$
   $F\,\Box_3(A \vee B)$

4. $T\,\Box_2 B$
   $F\,\Box_3(A \vee B)$

5. $T\,\Box_1 A$
   $F\,A \vee B$

6. $T\,\Box_2 B$
   $F\,A \vee B$

7. ~~$T\,\Box_1 A$~~
   $F\,A \vee B$
   $T\,A$

8. ~~$T\,\Box_2 B$~~
   $F\,A \vee B$
   $T\,B$

9. ~~$T\,\Box_1 A$~~
   $T\,A$
   $F\,A$
   $F\,B$

10. ~~$T\,\Box_2 B$~~
    $T\,B$
    $F\,A$
    $F\,B$

Figure 3.1: S4 Tableau Proof (to be expanded)

Next, the proof created in Figure 3.1 is converted to a mixed tableau, displayed in Figure 3.2. The work is from bottom up. In Figure 3.1, 9 is an atomically closed branch. For this, the algorithm makes use of a *trivial* expansion, giving the corresponding 9 of Figure 3.2, and similarly for 10.

Branch 7 in Figure 3.1 yields branch 9 by an α rule. Since 9 in Figure 3.1 expands to 9 in Figure 3.2, 7 of Figure 3.1 converts to 7 of Figure 3.2 by the α case of the Algorithm. Similarly for 8 and 10. Then branch 5 of Figure 3.1 converts to 5 of Figure 3.2 because of the 7 conversion, and the $T\,\Box$ case of the Algorithm, and similarly for 6 and 8. Branch 3 of Figure 3.1 yields branch 5 by the $F\Box$ rule. The associated justification formula for branch 5 is $v_1{:}A \supset (A \vee B)$. Justification term t, in 3, is such that $v_1{:}A \supset t{:}(A \vee B)$ is provable in LP. Existence is guaranteed by the Lifting Lemma 1.2.2. Similarly u in branch 4 is such that $v_2{:}B \supset u{:}(A \vee B)$. Branch 2 yields branches 3 and 4 using the β rule. Note that in branch 2 in Figure 3.2, the justification part associated with $F\,\Box_3(A \vee B)$ is the union of those parts from branches 3 and 5. Finally 1 is a straightforward application of the α rule.

Then, according to the algorithm, $\{F\,(v_1{:}A \vee v_2{:}B) \supset t{:}(A \vee B), F\,(v_1{:}A \vee v_2{:}B) \supset u{:}(A \vee B)\}$ is a set of quasi-realizers for $F\,(\Box_1 A \vee \Box_2 B) \supset \Box_3(A \vee B)$. We will see that

$$\bigvee\{(v_1{:}A \vee v_2{:}B) \supset t{:}(A \vee B), (v_1{:}A \vee v_2{:}B) \supset u{:}(A \vee B)\}$$

is provable in LP.

1. $(F\,(\Box_1 A \vee \Box_2 B) \supset \Box_3(A \vee B), \{F\,(v_1{:}A \vee v_2{:}B) \supset t{:}(A \vee B)$
   $F\,(v_1{:}A \vee v_2{:}B) \supset u{:}(A \vee B)\})$
2. $(T\,\Box_1 A \vee \Box_2 B, \{T\,v_1{:}A \vee v_2{:}B\})$
   $(F\,\Box_3(A \vee B), \{F\,t{:}(A \vee B), F\,u{:}(A \vee B)\})$

3. $(T\,\Box_1 A, \{T\,v_1{:}A\})$
   $(F\,\Box_3(A \vee B), \{F\,t{:}(A \vee B)\})$

4. $(T\,\Box_2 B, \{T\,v_2{:}B\})$
   $(F\,\Box_3(A \vee B), \{F\,u{:}(A \vee B)\})$

5. $(T\,\Box_1 A, \{T\,v_1{:}A\})$
   $(F\,A \vee B, \{F\,A \vee B\})$

6. $(T\,\Box_2 B, \{T\,v_2{:}B\})$
   $(F\,A \vee B, \{F\,A \vee B\})$

7. $(\cancel{T\,\Box_1 A}, \{T\,v_1{:}A\})$
   $(F\,A \vee B, \{F\,A \vee B\})$
   $(T\,A, \{T\,A\})$

8. $(\cancel{T\,\Box_2 B}, \{T\,v_2{:}B\})$
   $(F\,A \vee B, \{F\,A \vee B\})$
   $(T\,B, \{T\,B\})$

9. $(\cancel{T\,\Box_1 A}, \{T\,v_1{:}A\})$
   $(T\,A, \{T\,A\})$
   $(F\,A, \{F\,A\})$
   $(F\,B, \{F\,B\})$

10. $(\cancel{T\,\Box_2 B}, \{T\,v_2{:}B\})$
    $(T\,B, \{T\,B\})$
    $(F\,A, \{F\,A\})$
    $(F\,B, \{F\,B\})$

Justification term t, in 3, is such that $v_1{:}A \supset t{:}(A \vee B)$ is provable in LP. Similarly u in 4 is such that $v_2{:}B \supset u{:}(A \vee B)$ is LP provable.

Figure 3.2: S4 Tableau Proof (expanded)

3.5 Quasi-Realization Algorithm Correctness Proof

This section is devoted entirely to showing the correctness of Algorithm 3.4.1, and hence proving Theorem 3.3.3. It is straightforward that the algorithm produces a mixed tableau expansion. Details are left to the reader. We concentrate on showing the resulting mixed tableau must be justification sound, Definition 3.3.2. To do this, we show it for the Atomic Cases, and show that each rule of the algorithm preserves justification soundness.

Proof of correctness for Algorithm 3.4.1

Atomic Cases Consider the first of the two atomic cases—the second is similar. The mixed tableau branch produced in this case is $\mathcal{B}^E, T\,P, F\,P$. The associated justification formula is $[\bigwedge(\mathcal{B}^E)^{just}_T \wedge P] \supset [\bigvee(\mathcal{B}^E)^{just}_F \vee P]$, and this is trivially an LP theorem, so the branch is justification sound.

α Case Assume that $\mathcal{B}^E, (\alpha_1, S_1), (\alpha_2, S_2)$ is a mixed tableau branch that is justification sound. We must show the same for $\mathcal{B}^E, (\alpha, S)$ where $S = \{\alpha' \mid \alpha'_1 \in S_1$ and $\alpha'_2 \in S_2\}$. Since $S_1 \subseteq \langle\!\langle \alpha_1 \rangle\!\rangle$

and $S_2 \subseteq \langle\!\langle \alpha_2 \rangle\!\rangle$, it is easy to see from Definition 3.2.1 that $S \subseteq \langle\!\langle \alpha \rangle\!\rangle$. After this case we leave such arguments to the reader. We must show the branch is justification sound.

Since we only consider $\wedge$, $\vee$, and $\supset$, there are three possibilities for α. We look at one of them, with $\alpha = F\,A \supset B$; the other two cases are similar. All three could be condensed into a single argument by making use of uniform notation, but this would be a bit of a diversion just now. So, assume $\mathcal{B}^E, (T\,A, S_1), (F\,B, S_2)$ is justification sound; we show the same for $\mathcal{B}^E, (F\,A \supset B, S)$.

Let us say $S_1 = \{T\,A_1, \ldots, T\,A_m\}$ and $S_2 = \{F\,B_1, \ldots, F\,B_n\}$. Then the associated justification formula for $\mathcal{B}^E, (T\,A, S_1), (F\,B, S_2)$ is the following.

$$\Big[\bigwedge(\mathcal{B}^E)^{just}_T \wedge \bigwedge\{A_1, \ldots, A_m\}\Big] \supset \Big[\bigvee(\mathcal{B}^E)^{just}_F \vee \bigvee\{B_1, \ldots, B_n\}\Big]$$

By classical logic we also have provability of the following, where i ranges over $1, \ldots, m$ and j ranges over $1, \ldots, n$.

$$\bigwedge(\mathcal{B}^E)^{just}_T \supset \left[\bigvee(\mathcal{B}^E)^{just}_F \vee \bigvee_{i,j}(A_i \supset B_j)\right]$$

Thus the associated justification formula for $\mathcal{B}, (F\,A \supset B, S)$ is provable.

β **Case** Assume that $\mathcal{B}^{E_1}, (\beta_1, S_1)$ and $\mathcal{B}^{E_2}, (\beta_2, S_2)$ are justification sound. We show this also the case for $\beta^E, (\beta, S)$, where $S = \{\beta' \mid \beta'_1 \in S_1 \text{ and } \beta'_2 \in S_2\}$ and $\mathcal{B}^E = \mathcal{B}^{E_1} \dot{\cup} \mathcal{B}^{E_2}$. As with α there are three cases, and we only consider one of them, where $\beta = T\,A \supset B$. So, assume that $\mathcal{B}^{E_1}, (F\,A, S_1)$ and $\mathcal{B}^{E_2}, (T\,B, S_2)$ are justification sound.

Suppose $S_1 = \{F\,A_1, \ldots, F\,A_m\}$ and $S_2 = \{T\,B_1, \ldots, T\,B_n\}$. Then the provable associated justification formulas for $\mathcal{B}^{E_1}, (F\,A, S_1)$ and $\mathcal{B}^{E_2}, (T\,B, S_2)$ are the following.

$$\bigwedge(\mathcal{B}^{E_1})^{just}_T \supset \Big[\bigvee(\mathcal{B}^{E_1})^{just}_F \vee \bigvee\{A_1, \ldots, A_m\}\Big]$$

$$\Big[\bigwedge(\mathcal{B}^{E_2})^{just}_T \wedge \bigwedge\{B_1 \wedge \ldots \wedge B_n\}\Big] \supset \bigvee(\mathcal{B}^{E_2})^{just}_F$$

$\mathcal{B}^E = \mathcal{B}^{E_1} \dot{\cup} \mathcal{B}^{E_2}$, and it follows easily that $(\mathcal{B}^E)^{just}_T = (\mathcal{B}^{E_1})^{just}_T \cup (\mathcal{B}^{E_2})^{just}_T$ and $(\mathcal{B}^E)^{just}_F = (\mathcal{B}^{E_1})^{just}_F \cup (\mathcal{B}^{E_2})^{just}_F$. Then we have provability of the following.

$$\bigwedge(\mathcal{B}^E)^{just}_T \supset \Big[\bigvee(\mathcal{B}^E)^{just}_F \vee \bigvee\{A_1, \ldots, A_m\}\Big]$$

$$\Big[\bigwedge(\mathcal{B}^E)^{just}_T \wedge \bigwedge\{B_1 \wedge \ldots \wedge B_n\}\Big] \supset \bigvee(\mathcal{B}^E)^{just}_F$$

By classical logic this gives provability of the following, where i ranges over $1, 2, \ldots, m$ and j ranges over $1, 2, \ldots, n$.

$$\left[\bigwedge(\mathcal{B}^E)^{just}_T \wedge \bigwedge_{i,j}(A_i \supset B_j)\right] \supset \bigvee(\mathcal{B}^E)^{just}_F$$

Thus the associated justification formula for $\mathcal{B}, (T\,A \supset B, S)$ is provable.

Negation Cases These cases are similar to the α and β cases, but are simpler and are left to the reader.

$T\Box$ **Case** Assume that $\mathcal{B}^E, (\cancel{T\Box_k X}, S_0), (T\,X, S_1)$ is a justification sound mixed tableau branch. Then $\mathcal{B}^E, (T\,\Box_k X, S)$ is a mixed tableau branch, where $S = S_0 \cup \{T\,v_k{:}Z \mid T\,Z \in S_1\}$. We show it is justification sound.

Suppose $S_0 = \{T\,v_k : W_1, \ldots, T\,v_k : W_m\}$ and $S_1 = \{T\,Z_1, \ldots, T\,Z_k\}$. Then the provable associated justification formula for $\mathcal{B}^E, (\cancel{T\Box_k X}, S_0), (T\,X, S_1)$ is the following.

$$\Big[\bigwedge(\mathcal{B}^E)^{just}_T \wedge \bigwedge\{v_k{:}W_1, \ldots, v_k{:}W_m\} \wedge \bigwedge\{Z_1, \ldots, Z_k\}\Big] \supset \bigvee(\mathcal{B}^E)^{just}_F$$

Using Factivity, Axiom A2, we have LP provability of the following.

$$\Big[\bigwedge(\mathcal{B}^E)^{just}_T \wedge \bigwedge\{v_k{:}W_1, \ldots, v_k{:}W_m, v_k{:}Z_1, \ldots, v_k{:}Z_k\}\Big] \supset \bigvee(\mathcal{B}^E)^{just}_F$$

This is the associated justification formula for $\mathcal{B}^E, (T\,\Box_k X, S)$.

$F\Box$ **Case** Assume that $(\mathcal{B}^\sharp)^E, (F\,X, S_0)$ is a justification sound mixed tableau branch. Then $(\mathcal{B}^\sharp)^E, (\mathcal{B} - \mathcal{B}^\sharp)^E, (F\,\Box_n X, \{F\,t : \bigvee S\})$ is a mixed tableau branch, where $(\mathcal{B} - \mathcal{B}^\sharp)^E$ trivially expands $\mathcal{B} - \mathcal{B}^\sharp$, $S = \{Z \mid F\,Z \in S_0\}$, and t is any justification term. We show $(\mathcal{B}^\sharp)^E, (\mathcal{B} - \mathcal{B}^\sharp)^E, (F\,\Box_n X, \{F\,t{:}\bigvee S\})$ is justification sound, given the right choice of t.

Note that since all members of $\mathcal{B}^\sharp$ are T-signed, the LP-provable associated justification formula for $(\mathcal{B}^\sharp)^E, (F\,X, S_0)$ is simply $\bigwedge((\mathcal{B}^\sharp)^E)^{just}_T \supset \bigvee S$, where $S = \{Z \mid F\,Z \in S_0\}$. Also members of $\mathcal{B}^\sharp$ are necessitated, so by the Lifting Lemma 1.2.2, for some justification term t, $\vdash_{\mathsf{LP}} \bigwedge((\mathcal{B}^\sharp)^E)^{just}_T \supset t{:}\bigvee S$. Then, trivially, the following is also LP-provable

$$\Big[\bigwedge((\mathcal{B}^\sharp)^E)^{just}_T \wedge \bigwedge((\mathcal{B} - \mathcal{B}^\sharp)^E)^{just}_T\Big] \supset \Big[t{:}\bigvee S \vee \bigvee((\mathcal{B} - \mathcal{B}^\sharp)^E)^{just}_F\Big]$$

and this is the associated justification formula for $(\mathcal{B}^\sharp)^E, (\mathcal{B} - \mathcal{B}^\sharp)^E, (F\,\Box_n X, \{F\,t{:}\bigvee S\})$ as specified by the algorithm.

■

Chapter 4

Realizations

4.1 The Plan

In the previous chapter we gave an algorithm that constructs quasi-realizer sets, using cut-free tableau proofs as input. Now we give a second algorithm that converts quasi-realizer sets to realizations, which are single justification formulas. Tableau proofs play no role now, though signed formulas are still useful.

4.2 Realizations

We follow the structure of our characterization of quasi-realizations, Definition 3.2.1, but part of case 4 is different. Roughly speaking, the disjunction appearing in the definition of quasi-realizer will be folded into the justification term by using the + operator. We still assume that v_1, v_2, ... is an enumeration of all justification variables of LP, with no justification variable repeated.

Definition 4.2.1 The mapping $[\![\cdot]\!]$ is defined recursively on the set of signed annotated modal formulas.

1. If A is atomic, $[\![T\,A]\!] = \{T\,A\}$ and $[\![F\,A]\!] = \{F\,A\}$.

2. $[\![T\,\neg A]\!] = \{T\,\neg U \mid F\,U \in [\![F\,A]\!]\}$.
   $[\![F\,\neg A]\!] = \{F\,\neg U \mid T\,U \in [\![T\,A]\!]\}$.

3. $[\![\alpha]\!] = \{\alpha' \mid \alpha'_1 \in [\![\alpha_1]\!]$ and $\alpha'_2 \in [\![\alpha_2]\!]\}$.
   $[\![\beta]\!] = \{\beta' \mid \beta'_1 \in [\![\beta_1]\!]$ and $\beta'_2 \in [\![\beta_2]\!]\}$.

4. $[\![T\,\Box_n A]\!] = \{T\,v_n{:}U \mid T\,U \in [\![T\,A]\!]\}$.
   $[\![F\,\Box_n A]\!] = \{F\,t{:}U \mid F\,U \in [\![F\,A]\!]$ and t is any justification term$\}$.

5. The mapping is extended to *sets* of signed annotated formulas by letting $[\![S]\!] = \cup\{[\![Z]\!] \mid Z \in S\}$.

Members of $[\![Z]\!]$ are called *realizers* of Z, where Z is a signed, annotated modal formula. In particular, a *normal realization* of annotated modal A is any justification formula U where $F\,U \in [\![F\,A]\!]$. For a modal formula A without annotations, a normal realization for A is any normal realization for A', where A' is an annotated version of A.

It should be noted that we are not requiring realizers to be provable. We considered using the term *potential realizer*, but found it too unwieldy. Of course, what we are after is a *provable realizer* for each S4 theorem.

4.3 Substitution

Substitutions play an essential role in this Chapter. They replace justification variables with justification terms. A substitution is a function σ, typically denoted $\{v_{i_1}/t_1, \ldots, v_{i_n}/t_n\}$, mapping justification variable v_{i_k} to justification term t_k, and is the identity otherwise (it is assumed that each t_i is different from v_{i_k}). The *domain* of σ is $\{v_{i_1}, \ldots, v_{i_n}\}$. For a justification formula A the result of applying a substitution σ is denoted $A\sigma$; likewise $t\sigma$ is the result of applying substitution σ to justification term t.

Substitutions turn LP theorems into LP theorems, because they turn axioms into axioms (axiomatization is by schemes), and rule applications into rule applications. Still, the role of constants changes with a substitution. Suppose $\mathcal{C}$ is a constant specification, A is an axiom, and $c{:}A$ is added to a proof using Axiom Necessitation, where this addition meets constant specification $\mathcal{C}$. Since $A\sigma$ is also an axiom, Axiom Necessitation allows us to add $c{:}A\sigma$ to a proof, but this may no longer meet specification $\mathcal{C}$. A new constant specification, call it $\mathcal{C}\sigma$, can be computed from the original one—$c{:}A\sigma \in \mathcal{C}\sigma$ just in case $c{:}A \in \mathcal{C}$. If $\mathcal{C}$ was axiomatically appropriate, $\mathcal{C} \cup \mathcal{C}\sigma$ will also be. So, if A is provable using an axiomatically appropriate constant specification the same will be true for $A\sigma$. From now on we skip such details.

Definition 4.3.1 Let σ be a substitution, and A be an annotated modal formula.

1. σ *lives on* A if, for every justification variable v_k in the domain of σ, $\Box_k$ occurs in A;
2. σ *lives away from* A if, for every justification variable v_k in the domain of σ, $\Box_k$ does not occur in A;
3. σ meets the *no new variable* condition if, for every v_k in the domain of σ, the justification term $v_k\sigma$ contains no variables other than v_k.

Proposition 4.3.2 *Assume A is an annotated modal formula, σ_A is a substitution that lives on A, and σ_Z is a substitution that lives away from A.*

1. *If $T\,U \in [\![T\,A]\!]$ then $T\,U\sigma_Z \in [\![T\,A]\!]$; if $F\,U \in [\![F\,A]\!]$ then $F\,U\sigma_Z \in [\![F\,A]\!]$*
2. *Further, if both σ_A and σ_Z meet the no new variable condition, then $\sigma_A\sigma_Z = \sigma_Z\sigma_A$.*

Proof Part 1: The proof is by induction on the complexity of A. The atomic case is trivial since no justification variables are present, and the propositional cases are straightforward. This leaves the modal cases. Suppose $A = \Box_n B$, and the result is known for simpler formulas.

Assume that $T\,v_n{:}U \in [\![T\,\Box_n B]\!]$. Since σ_Z lives away from A, $v_n\sigma_Z = v_n$. By the induction hypothesis $T\,U\sigma_Z \in [\![T\,B]\!]$. Then $T\,(v_n{:}U)\sigma_Z = T\,v_n{:}(U\sigma_Z) \in [\![T\,\Box_n B]\!]$.

Assume $F\,t{:}U \in [\![F\,\Box_n B]\!]$. By the induction hypothesis, $F\,U\sigma_Z \in [\![F\,B]\!]$. Then $F\,(t{:}U)\sigma_Z = F\,t\sigma_Z{:}U\sigma_Z \in [\![F\,\Box_n B]\!]$

Part 2: Assume the hypothesis, and let v_k be a justification variable; we show $v_k\sigma_A\sigma_Z = v_k\sigma_Z\sigma_A$.

First, suppose $\Box_k$ occurs in A. Since σ_A meets the no new variable condition, the only justification variable that can occur in $v_k\sigma_A$ is v_k. Since σ_Z lives away from A, $v_k\sigma_Z = v_k$, and so $v_k\sigma_A\sigma_Z = v_k\sigma_A$. But also, $v_k\sigma_Z\sigma_A = v_k\sigma_A$, hence $v_k\sigma_A\sigma_Z = v_k\sigma_Z\sigma_A$.

Second, suppose $\Box_k$ does not occur in A. Since σ_A lives on A, $v_k\sigma_A = v_k$. And since σ_Z meets the no new variable condition, v_k is the only variable that can occur in $v_k\sigma_Z$. Then $v_k\sigma_Z\sigma_A = v_k\sigma_Z$, and $v_k\sigma_A\sigma_Z = v_k\sigma_Z$, so $v_k\sigma_A\sigma_Z = v_k\sigma_Z\sigma_A$. ■

4.4 The Quasi-Realization to Realization Algorithm

In this section we give an algorithm for *condensing* a quasi-realization set to a single realizer. Just as we did with our Quasi-Realization Construction algorithm, we introduce some special notation to make the algorithm below more easily presentable.

Definition 4.4.1 (Condensing) Let A be an annotated modal formula, $\mathcal{A}$ be a set of justification formulas, A' be a single justification formula, and σ be a substitution.

1. $\mathcal{A} \xrightarrow{T\,A} (A', \sigma)$ means: σ lives on A and meets the no new variable condition; $T\,\mathcal{A} \subseteq \langle\!\langle T\,A \rangle\!\rangle$; $T\,A' \in [\![T\,A]\!]$; and $\vdash_{\mathsf{LP}} A' \supset (\bigwedge \mathcal{A})\sigma$.

2. $\mathcal{A} \xrightarrow{F\,A} (A', \sigma)$ means: σ lives on A and meets the no new variable condition; $F\,\mathcal{A} \subseteq \langle\!\langle F\,A \rangle\!\rangle$; $F\,A' \in [\![F\,A]\!]$; and $\vdash_{\mathsf{LP}} (\bigvee \mathcal{A})\sigma \supset A'$.

One can read the notation $\mathcal{A} \xrightarrow{T\,A} (A', \sigma)$ as saying that the set of quasi-realizers $\mathcal{A}$ for $T\,A$ *condenses* to the single realizer $T\,A'$ using substitution σ, and similarly for $\mathcal{A} \xrightarrow{F\,A} (A', \sigma)$.

Our algorithm provides a constructive proof of the following, which in turn will give us LP provable realizations of $\mathsf{S4}$ provable modal formulas in the next section.

Theorem 4.4.2 *Let A be an annotated modal formula. For each finite set $\mathcal{A}$ of justification formulas:*

1. *If $T\,\mathcal{A} \subseteq \langle\!\langle T\,A \rangle\!\rangle$ then there are A' and σ so that $\mathcal{A} \xrightarrow{T\,A} (A', \sigma)$.*

2. *If $F\,\mathcal{A} \subseteq \langle\!\langle F\,A \rangle\!\rangle$ then there are A' and σ so that $\mathcal{A} \xrightarrow{F\,A} (A', \sigma)$.*

The algorithm below proceeds by induction on the formula complexity of X. The atomic case is simple. For the other cases we give a construction that makes use of the notation introduced above. In each case, if the schemes above the line are the case, so is the scheme below. Uniform notation is not helpful now. A correctness proof for the algorithm is in Section 4.5.

Algorithm 4.4.3 (Quasi-Realization to Realization Condensing)

Atomic Cases Trivial, since if P is atomic $\langle\!\langle P \rangle\!\rangle = [\![P]\!] = \{P\}$, and we can use the empty substitution, ϵ. So we have the following.

$$\{P\} \xrightarrow{T\,P} (P, \epsilon) \qquad\qquad \{P\} \xrightarrow{F\,P} (P, \epsilon)$$

T $\supset$ Case

$$\frac{\{A_1, \ldots, A_k\} \xrightarrow{F\,A} (A', \sigma_A) \qquad \{B_1, \ldots, B_k\} \xrightarrow{T\,B} (B', \sigma_B)}{\{A_1 \supset B_1, \ldots, A_k \supset B_k\} \xrightarrow{T\,A \supset B} (A'\sigma_B \supset B'\sigma_A, \sigma_A\sigma_B)}$$

F $\supset$ Case

$$\frac{\{A_1, \ldots, A_k\} \xrightarrow{T\,A} (A', \sigma_A) \qquad \{B_1, \ldots, B_k\} \xrightarrow{F\,B} (B', \sigma_B)}{\{A_1 \supset B_1, \ldots, A_k \supset B_k\} \xrightarrow{F\,A \supset B} (A'\sigma_B \supset B'\sigma_A, \sigma_A\sigma_B)}$$

T $\wedge$ Case

$$\frac{\{A_1,\ldots,A_k\} \xrightarrow{T\,A} (A',\sigma_A) \qquad \{B_1,\ldots,B_k\} \xrightarrow{T\,B} (B',\sigma_B)}{\{A_1 \wedge B_1,\ldots,A_k \wedge B_k\} \xrightarrow{T\,A\wedge B} (A'\sigma_B \wedge B'\sigma_A, \sigma_A\sigma_B)}$$

F $\wedge$ Case

$$\frac{\{A_1,\ldots,A_k\} \xrightarrow{F\,A} (A',\sigma_A) \qquad \{B_1,\ldots,B_k\} \xrightarrow{F\,B} (B',\sigma_B)}{\{A_1 \wedge B_1,\ldots,A_k \wedge B_k\} \xrightarrow{F\,A\wedge B} (A'\sigma_B \wedge B'\sigma_A, \sigma_A\sigma_B)}$$

T $\vee$ Case

$$\frac{\{A_1,\ldots,A_k\} \xrightarrow{T\,A} (A',\sigma_A) \qquad \{B_1,\ldots,B_k\} \xrightarrow{T\,B} (B',\sigma_B)}{\{A_1 \vee B_1,\ldots,A_k \vee B_k\} \xrightarrow{T\,A\vee B} (A'\sigma_B \vee B'\sigma_A, \sigma_A\sigma_B)}$$

F $\vee$ Case

$$\frac{\{A_1,\ldots,A_k\} \xrightarrow{F\,A} (A',\sigma_A) \qquad \{B_1,\ldots,B_k\} \xrightarrow{F\,B} (B',\sigma_B)}{\{A_1 \vee B_1,\ldots,A_k \vee B_k\} \xrightarrow{F\,A\vee B} (A'\sigma_B \vee B'\sigma_A, \sigma_A\sigma_B)}$$

T $\neg$ Case

$$\frac{\{A_1,\ldots,A_k\} \xrightarrow{F\,A} (A',\sigma_A)}{\{\neg A_1,\ldots,\neg A_k\} \xrightarrow{T\,\neg A} (\neg A',\sigma_A)}$$

F $\neg$ Case

$$\frac{\{A_1,\ldots,A_k\} \xrightarrow{T\,A} (A',\sigma_A)}{\{\neg A_1,\ldots,\neg A_k\} \xrightarrow{F\,\neg A} (\neg A',\sigma_A)}$$

T $\Box$ Case

$$\frac{\{A_1,\ldots,A_k\} \xrightarrow{T\,A} (A',\sigma_A)}{\{v_n{:}A_1,\ldots,v_n{:}A_k\} \xrightarrow{T\,\Box_n A} (v_n{:}A'\sigma, \sigma_A\sigma)}$$

where $\vdash_{\mathsf{LP}} t_i{:}(A' \supset A_i\sigma_A)$
for $i = 1,\ldots,k$,
$s = t_1 + \ldots + t_k$
$\sigma = \{v_n/(s \cdot v_n)\}$

F $\Box$ Case

$$\frac{\mathcal{A}_1 \cup \ldots \cup \mathcal{A}_k \xrightarrow{F\,A} (A',\sigma_A)}{\{t_1{:}\bigvee \mathcal{A}_1,\ldots,t_k{:}\bigvee \mathcal{A}_k\} \xrightarrow{F\,\Box_n A} (t\sigma_A{:}A',\sigma_A)}$$

where $\vdash_{\mathsf{LP}} u_i{:}(\bigvee \mathcal{A}_i\sigma_A \supset A')$
for $i = 1,\ldots,k$,
$t = u_1 \cdot t_1 + \ldots + u_k \cdot t_k$

At the end of Section 3.4 we presented an example showing that the annotated modal formula $(\Box_1 A \vee \Box_2 B) \supset \Box_3(A \vee B)$, provable in (annotated) $\mathsf{S4}$, has the following quasi-realization set,

$$\{(v_1{:}A \vee v_2{:}B) \supset t{:}(A \vee B), (v_1{:}A \vee v_2{:}B) \supset u{:}(A \vee B)\}$$

where $\vdash_{\mathsf{LP}} v_1{:}A \supset t{:}(A \vee B)$ and $\vdash_{\mathsf{LP}} v_2{:}B \supset u{:}(A \vee B)$. Then the disjunction

$$[(v_1{:}A \vee v_2{:}B) \supset t{:}(A \vee B)] \vee [(v_1{:}A \vee v_2{:}B) \supset u{:}(A \vee B)]$$

is provable in LP. Now we can apply Algorithm 4.4.3, Quasi-Realization to Realization Condensing. We leave the details to you—the final stage is the following.

$$\begin{array}{l}\{(v_1{:}A \vee v_2{:}B) \supset t{:}(A \vee B),\\ (v_1{:}A \vee v_2{:}B) \supset u{:}(A \vee B)\}\end{array} \xrightarrow{F\,(\Box_1 A \vee \Box_2 B) \supset \Box_3 (A \vee B)} ((v_1{:}A \vee v_2{:}B) \supset ((c \cdot t + c \cdot u){:}(A \vee B))\sigma_0, \sigma_0)$$

In this a internalizes a proof of $A \supset A$, b internalizes a proof of $B \supset B$, c internalizes a proof of $(A \vee B) \supset (A \vee B)$, and σ_0 is the substitution $\{v_1/a \cdot v_1, v_2/b \cdot v_2\}$. Since A and B are atomic, and c contains no justification variables, this can be rewritten as the following.

$$\begin{array}{l}\{(v_1{:}A \vee v_2{:}B) \supset t{:}(A \vee B),\\ (v_1{:}A \vee v_2{:}B) \supset u{:}(A \vee B)\}\end{array} \xrightarrow{F\,(\Box_1 A \vee \Box_2 B) \supset \Box_3 (A \vee B)} ((v_1{:}A \vee v_2{:}B) \supset (c \cdot t\sigma_0 + c \cdot u\sigma_0){:}(A \vee B), \sigma_0)$$

In fact, $((v_1{:}A \vee v_2{:}B) \supset (c \cdot t\sigma_0 + c \cdot u\sigma_0){:}(A \vee B), \sigma_0)$ is a normal realization of $(\Box_1 A \vee \Box_2 B) \supset \Box_3 (A \vee B)$, and Theorem 4.6.1 will show that it is provable in LP.

4.5 Correctness Proof for the Condensing Algorithm

Proof of correctness for Algorithm 4.4.3 (Showing correctness of the Algorithm serves to establish Theorem 4.4.2.) We must justify each case of the algorithm. We give two propositional cases in considerable detail and abbreviate or omit the rest. The modal cases are fully presented.

Atomic Cases These cases are immediate.

T $\supset$ Case Assume we are given $\{T\,A_1 \supset B_1, \ldots, T\,A_k \supset B_k\} \subseteq \langle\!\langle T\,A \supset B \rangle\!\rangle$, and also we have the following.

$$\{A_1, \ldots, A_k\} \xrightarrow{F\,A} (A', \sigma_A) \qquad \{B_1, \ldots, B_k\} \xrightarrow{T\,B} (B', \sigma_B)$$

Then we also have that $\{F\,A_1, \ldots, F\,A_k\} \subseteq \langle\!\langle F\,A \rangle\!\rangle$ and $\{T\,B_1, \ldots, T\,B_k\} \subseteq \langle\!\langle T\,B \rangle\!\rangle$. Since $\{A_1, \ldots, A_k\} \xrightarrow{F\,A} (A', \sigma_A)$ we have A' and σ_A so that σ_A lives on A and meets the no new variable condition, $F\,A' \in [\![F\,A]\!]$, and $(A_1 \vee \ldots \vee A_k)\sigma_A \supset A'$ is provable in LP. Similarly, since $\{B_1, \ldots, B_k\} \xrightarrow{T\,B} (B', \sigma_B)$ we have B' and σ_B so that σ_B lives on B and meets the no new variable condition, $T\,B' \in [\![T\,B]\!]$, and $B' \supset (B_1 \wedge \ldots \wedge B_k)\sigma_B$ is provable in LP.

Since $A \supset B$ is an annotated modal formula then A and B have no indexes in common, because indexes can appear only once in a formula. Then σ_A and σ_B have disjoint domains. In particular, σ_A lives on A and so lives away from B, while σ_B lives on B and so lives away from A. Then $\sigma_A\sigma_B = \sigma_B\sigma_A$ by Proposition 4.3.2. It is easy to see that $\sigma_A\sigma_B$ lives on $A \supset B$ and meets the no new variable condition.

Again by Proposition 4.3.2, $F\,A'\sigma_B \in [\![F\,A]\!]$ since $F\,A' \in [\![F\,A]\!]$ and σ_B lives away from A. Likewise $T\,B'\sigma_A \in [\![T\,B]\!]$. Then $T\,A'\sigma_B \supset B'\sigma_A \in [\![T\,A \supset B]\!]$.

Finally, since $(A_1 \vee \ldots \vee A_k)\sigma_A \supset A'$ is provable in LP, so is $[(A_1 \vee \ldots \vee A_k)\sigma_A \supset A']\sigma_B = (A_1 \vee \ldots \vee A_k)\sigma_A\sigma_B \supset A'\sigma_B$. Similarly $B'\sigma_A \supset (B_1 \wedge \ldots \wedge B_k)\sigma_B\sigma_A$ is provable, or equivalently, $B'\sigma_A \supset (B_1 \wedge \ldots \wedge B_k)\sigma_A\sigma_B$. Then by classical logic, the following is LP provable.

$$(A'\sigma_B \supset B'\sigma_A) \supset [(A_1 \supset B_1) \vee \ldots \vee (A_k \supset B_k)]\sigma_A\sigma_B$$

We have now established the following, completing the case.

$$\{A_1 \supset B_1, \ldots, A_k \supset B_k\} \xrightarrow{T\,A \supset B} (A'\sigma_B \supset B'\sigma_A, \sigma_A\sigma_B)$$

F $\supset$ Case Assume we are given $\{F\,A_1 \supset B_1, \ldots, F\,A_k \supset B_k\} \subseteq \langle\!\langle F\,A \supset B \rangle\!\rangle$, and we also have the following.

$$\{A_1, \ldots, A_k\} \xrightarrow{T\,A} (A', \sigma_A) \qquad\qquad \{B_1, \ldots, B_k\} \xrightarrow{F\,B} (B', \sigma_B)$$

As in the previous case, $\sigma_A\sigma_B = \sigma_B\sigma_A$. Also $T\,A'\sigma_B \in [\![T\,A]\!]$ and $F\,B'\sigma_A \in [\![F\,B]\!]$, so $F\,A'\sigma_B \supset B'\sigma_A \in [\![F\,A \supset B]\!]$. Likewise the following is provable in LP.

$$(A'\sigma_B \supset B'\sigma_A) \supset [(A_1 \supset B_1) \wedge \ldots \wedge (A_k \supset B_k)]\sigma_A\sigma_B$$

All this establishes the following.

$$\{A_1 \supset B_1, \ldots, A_k \supset B_k\} \xrightarrow{F\,A \supset B} (A'\sigma_B \supset B'\sigma_A, \sigma_A\sigma_B)$$

T $\wedge$, F $\wedge$, T $\vee$, F $\vee$, T $\neg$, F $\neg$ Cases These are similar to the implication cases, and are left to the reader.

T $\Box$ Case Assume we are given $\{T\,v_n{:}A_1, \ldots, T\,v_n{:}A_k\} \subseteq \langle\!\langle T\,\Box_n A \rangle\!\rangle$, and also the following.

$$\{A_1, \ldots, A_k\} \xrightarrow{T\,A} (A', \sigma_A)$$

From the first assumption, $\{T\,A_1, \ldots, T\,A_k\} \subseteq \langle\!\langle T\,A \rangle\!\rangle$. Then by the second assumption, there are σ_A and $T\,A' \in [\![T\,A]\!]$ such that $A' \supset (A_1 \wedge \ldots \wedge A_k)\sigma_A$ is LP provable, where σ_A lives on A and meets the no new variable condition.

For each $i = 1, \ldots, k$, the formula $A' \supset A_i\sigma_A$ is provable, so by the Lifting Lemma there is a justification term t_i (with no justification variables) such that $t_i{:}(A' \supset A_i\sigma_A)$ is LP provable. Let s be the justification term $t_1 + \ldots + t_k$. Then $s{:}(A' \supset A_i\sigma_A)$ is provable, for each i.

Let σ be the substitution $\{v_n/(s \cdot v_n)\}$. For each $i = 1, \ldots, k$, $s{:}(A' \supset A_i\sigma_A)$ is provable, hence so is $[s{:}(A' \supset A_i\sigma_A)]\sigma$. Since s is a justification term with no justification variables, $s{:}(A'\sigma \supset A_i\sigma_A\sigma)$ is provable. Then for each i, $v_n{:}A'\sigma \supset (s \cdot v_n){:}A_i(\sigma_A\sigma)$ is provable. Since $\Box_n A$ is an annotated modal formula indexes cannot occur more than once, hence index n cannot occur in A. Substitution σ_A lives on A, hence v_n is not in its domain. It follows that $v_n(\sigma_A\sigma) = v_n\sigma = (s \cdot v_n)$, and so $[v_n{:}A_i](\sigma_A\sigma) = (s \cdot v_n){:}A_i(\sigma_A\sigma)$. Then for each i, $v_n{:}A'\sigma \supset [v_n{:}A_i](\sigma_A\sigma)$ is provable, and so $v_n{:}A'\sigma \supset [v_n{:}A_1 \wedge \ldots \wedge v_n{:}A_k](\sigma_A\sigma)$ is provable.

The substitution σ lives away from A so, since $T\,A' \in [\![T\,A]\!]$ then also $T\,A'\sigma \in [\![T\,A]\!]$ by Proposition 4.3.2. Then $T\,v_n{:}A'\sigma \in [\![T\,\Box_n A]\!]$.

Finally, it is easy to check that $\sigma_A\sigma$ lives on $\Box_n A$ and meets the no new variable condition.

This is enough to establish the following.

$$\{v_n{:}A_1, \ldots, v_n{:}A_k\} \xrightarrow{T\,\Box_n A} (v_n{:}A'\sigma, \sigma_A\sigma)$$

F $\Box$ Case Assume we are given $\{F\,t_1{:}\bigvee \mathcal{A}_1, \ldots, F\,t_k{:}\bigvee \mathcal{A}_k\} \subseteq \langle\!\langle F\,\Box_n A \rangle\!\rangle$, and also the following.

$$(\mathcal{A}_1 \cup \ldots \cup \mathcal{A}_k) \xrightarrow{F\,A} (A', \sigma_A)$$

From the first assumption, $F\,\mathcal{A}_1 \cup \ldots \cup \mathcal{A}_k \subseteq \langle\!\langle F\,A \rangle\!\rangle$. Then by the second assumption there are σ_A and $F\,A' \in [\![F\,A]\!]$ such that $\bigvee\{\mathcal{A}_1 \cup \ldots \cup \mathcal{A}_k\}\sigma_A \supset A'$ is LP provable, where σ_A lives on A and meets the no new variable condition.

For each i, $\bigvee \mathcal{A}_i\sigma_A \supset A'$ is provable, so by the Lifting Lemma there is a justification term u_i (with no justification variables) such that $u_i{:}(\bigvee \mathcal{A}_i\sigma_A \supset A')$ is LP provable. Then $(t_i\sigma_A){:}\ \bigvee \mathcal{A}_i\sigma_A \supset (u_i \cdot (t_i\sigma_A)){:}A'$ is also provable, or equivalently, $(t_i{:}\bigvee \mathcal{A}_i)\sigma_A \supset (u_i \cdot (t_i\sigma_A)){:}A'$. Since u_i contains no justification variables, this in turn is equivalent to $(t_i{:}\bigvee \mathcal{A}_i)\sigma_A \supset ((u_i \cdot t_i)\sigma_A){:}A'$. Now let $t = u_1 \cdot t_1 + \ldots + u_k \cdot t_k$. Then for each i, $(t_i{:}\bigvee \mathcal{A}_i)\sigma_A \supset (t\sigma_A){:}A'$ is LP provable. This gives us LP provability of the following.

$$\left[t_1{:}\bigvee \mathcal{A}_1 \vee \ldots \vee t_k{:}\bigvee \mathcal{A}_k\right] \sigma_A \supset (t\sigma_A){:}A'$$

It is immediate that $F\,(t\sigma_A){:}A' \in [\![F\,\Box_n A]\!]$, and that σ_A lives on $\Box_n A$. We already know it meets the no new variable condition. We have thus verified the following.

$$\{t_1{:}\bigvee \mathcal{A}_1, \ldots, t_k{:}\bigvee \mathcal{A}_k\} \xrightarrow{F\,\Box_n A} (t\sigma_A{:}A', \sigma_A)$$

■

The construction and proof above trace back to Proposition 7.8 in [5], with a modification and correction supplied in [7].

4.6 Finishing Up

Theorem 4.6.1 (Realization) *Every formula provable in $\mathsf{S4}$ has a normal realization that is provable in LP.*

Proof Suppose X is a theorem of $\mathsf{S4}$. Let A be an annotated version of X, any one will do. Then from Theorem 3.2.3, proved using Algorithm 3.4.1, there are $Q_1, \ldots, Q_k$ with $\{F\,Q_1, \ldots, F\,Q_k\} \subseteq \langle\!\langle F\,A \rangle\!\rangle$ such that $Q_1 \vee \ldots \vee Q_k$ is a theorem of LP. By part 2 of Theorem 4.4.2, proved using Algorithm 4.4.3 there is a substitution σ and a formula A' with $F\,A' \in [\![F\,A]\!]$ such that $(Q_1 \vee \ldots \vee Q_k)\sigma \supset A'$ is a theorem of LP. Since $(Q_1 \vee \ldots \vee Q_k)\sigma$ must also be provable in LP, so is A', and this is a normal realization of A, and hence of X. ■

Chapter 5

An Implementation

5.1 The Program

In this section we give a program for computing realizers of S4 theorems, written in SWI Prolog. Examples of its execution are in Section 5.2, and a discussion of implementation details is in Section 5.5. The program is divided into parts that are marked out with lines of asterisks, and each part has substantial commentary.

```
/*  Realization constructor for propositional S4 into LP.  It is a combination
    of a quasi-realization program for S4 and a quasi-realization to
    realization program.  It begins by constructing a (destructive) tableau
    proof.  Then it fills that out, from branch end up, to make a mixed
    tableau, from which a quasi-realization is extracted.  Then this is
    converted into a realization.  Information is displayed for both
    quasi-realizations and realizations.

    The tableau construction uses signed formulas.  These are implemented
    simply as f(X) and t(X).  There is one more sign in use, u(X).
    Semantically, this is the same as t(X).  But in order to avoid infinite
    looping during tableau proof discovery, when a nu rule has been applied
    to t(X), it is changed to u(X), in effect to check it off as having been used.
    Further nu rule applications do not apply to u(X).  When a pi rule is applied,
    occurrences of u(X) are changed back to t(X) since, in a sense, the branch
    starts over at this point.  In fact, u(X) corresponds to t(X) crossed out,
    as in the accompanying tech report.

    Propositional operators are: neg, and, or, imp.
    The binary connectives are right associative.  Box is the only modal
    operator.  There is also a dedicated constant, bot (falsehood).
    In addition there is ::, for the "justifies" predicate (: was already
    in use).  It is right associative.  And there is * for the dot,
    or application, operator, and + as usual.  Both are left associative.

    Parentheses are recommended.
```

```
    When reading output, justification variables are v1, v2, etc.
    Justification terms, introduced at the quasi-realization stage,
    are j1, j2, etc.  Those introduced at the realization stage
    are t1, t2, etc.

    The implementation uses the swipl library, specifically for its
    member, union, and subtract predicates.

    March, 2013

    Melvin Fitting
*/

/*****************************************************************/

/* SYNTAX */

:-op(140, fy, [neg, box]).
:-op(160, xfy, [and, or, imp, ::]).
:-op(160, yfx, [*]).

/*
  Uniform notation for propositional  connectives is used.
  Since only box is permitted as a modal operator, modal
  uniform notation is not used.
*/

/*  conjunctive(X) :- X is a signed alpha formula.
*/

conjunctive(t(_ and _)).
conjunctive(f(_ or _)).
conjunctive(f(_ imp _)).

/*  disjunctive(X) :- X is a signed beta formula.
*/

disjunctive(f(_ and _)).
disjunctive(t(_ or _)).
disjunctive(t(_ imp _)).

/*  unary(X) :- X is a signed negation.
*/

unary(f(neg _)).
unary(t(neg _)).

/*  components(X, Y, Z) :- Y and Z are the components
```

```
      of the signed formula X, as defined in the alpha
      and beta tables.
*/

components(t(X and Y), t(X), t(Y)).
components(f(X and Y), f(X), f(Y)).
components(t(X or Y), t(X), t(Y)).
components(f(X or Y), f(X), f(Y)).
components(t(X imp Y), f(X), t(Y)).
components(f(X imp Y), t(X), f(Y)).

/*  component(X, Y) :- Y is the component of the
      unary formula X.
*/

component(f(neg X), t(X)).
component(t(neg X), f(X)).

/* propAtom(X) :- X is a signed propositional letter.
*/

propAtom(t(X)) :- atom(X).
propAtom(f(X)) :- atom(X).

/*  Justification variables will be v1, v2, etc, and justification
    terms will be j1, j2, etc, and t1, t2, etc.  These are generated
    using gensym. The following resets the counters.  Also, what each
    justification term serves to justify is recorded using
    assert, so that a table of values can be displayed at the end.
    The reset instruction also erases earlier results.
*/

reset :-
   reset_gensym(v),
   reset_gensym(j),
   reset_gensym(t),
   retractall(proves(_, _)),
   retractall(where(_,_,_,_,_)).

/*  Formulas must be annotated, with indexed occurrences of box.
    Eventually positive occurrences of indexed boxes must be replaced
    with justification variables.  It is simplest to just replace
    all box occurrences with justification variables up front, and treat
    them as if they were indexed boxes during the tableau construction.
    This avoids some complexity later on.  So, in the tableau
    construction below, box P will actually appear as vn::P, where
    vn is a justification variable.
```

```
    annotate(X, Y) :- X is a signed formula and Y is the result
    of replacing each occurrence of box in X with an occurrence
    of a new justification variable vn, so that Y is a signed
    formula of justification logic.
*/

annotate(X, X) :-
  propAtom(X).

annotate(Binary, NewBinary) :-
  conjunctive(Binary),
  components(Binary, Binaryone, Binarytwo),
  annotate(Binaryone, NewBinaryone),
  annotate(Binarytwo, NewBinarytwo),
  conjunctive(NewBinary),
  components(NewBinary, NewBinaryone, NewBinarytwo).

annotate(Binary, NewBinary) :-
  disjunctive(Binary),
  components(Binary, BinaryOne, BinaryTwo),
  annotate(BinaryOne, NewBinaryOne),
  annotate(BinaryTwo, NewBinaryTwo),
  disjunctive(NewBinary),
  components(NewBinary, NewBinaryOne, NewBinaryTwo).

annotate(Negation, NewNegation) :-
  unary(Negation),
  component(Negation, Body),
  annotate(Body, NewBody),
  component(NewNegation, NewBody).

annotate(t(box Body), t(Var :: NewBody)) :-
  annotate(t(Body), t(NewBody)),
  gensym(v, Var).

annotate(f(box Body), f(Var :: NewBody)) :-
  annotate(f(Body), f(NewBody)),
  gensym(v, Var).

/*************************************************************/

/* TABLEAU CONSTRUCTION */

/*  closes(MixedTableau, D) :- MixedTableau can be expanded to one whose
    modal part is closed, using modal depth D, a limit on the number of
    pi rule applications.  Then the justification part is filled in,
    branch end to root, to produce a proper closed mixed tableau.  If
    closure is not possible, a call fails.
```

```
    A mixed tableau is a list of its branches.  A branch is a list
    of the items on it.  Each item is of the form mixed(A, B), where
    A is a signed modal formula and B is a list of signed justification
    formulas.  Although lists are used to represent branches,
    they are treated as sets, and all repetitions are removed.

    Closure is always atomic closure.  Then the trivial(Branch) operation
    in the atomic closure clauses below provides justification counterparts
    for all formulas on the branch, including the one or ones that caused
    the branch to close.  Here we make use of the fact that,
    even in modal formulas, indexed box operators have been represented
    by justification variables.

    The order of these clauses is important.  If a tableau proof can be
    found, any ordering will serve, but some will find more complex proofs
    than others, with useless steps.  This will be reflected in the
    complexity of the quasi-realizers.
*/

/* ATOMIC */

closes([Branch | Rest], D) :-
  member(mixed(t(bot), _), Branch),
  trivial(Branch),
  closes(Rest, D).

closes([Branch | Rest], D) :-
  member(mixed(t(X), _), Branch),
  member(mixed(f(X), _), Branch),
  propAtom(t(X)),
  trivial(Branch),
  closes(Rest, D).

/* PI RULE */

closes([Branch | Rest], D) :-
  member(mixed(f(Var :: Fmla), [f(Term :: Disjunction)]), Branch),
  D > 1,
  subtract(Branch, [mixed(f(Var :: Fmla), [f(Term :: Disjunction)])],
    ReducedBranch), sharp(ReducedBranch, Temporary),
  union([mixed(f(Fmla), JList)], Temporary, NewBranch),
  NewD is D - 1,
  closes([NewBranch | Rest], NewD),
  disjunctionOfList(JList, Disjunction),
  trivial(ReducedBranch),
  gensym(j, Term),
  assert(proves(Term, NewBranch)).
```

```
/* UNARY RULE */

closes([Branch | Rest], D) :-
  member(mixed(Negation, JNegatedList), Branch),
  unary(Negation),
  component(Negation, UnNegated),
  subtract(Branch, [mixed(Negation, JNegatedList)], Temporary),
  union([mixed(UnNegated, JUnNegatedList)], Temporary, NewBranch),
  closes([NewBranch | Rest], D),
  negate(JUnNegatedList, JNegatedList).

/* ALPHA RULE */

closes([Branch | Rest], D) :-
  member(mixed(Alpha, JAlphaList), Branch),
  conjunctive(Alpha),
  components(Alpha, AlphaOne, AlphaTwo),
  subtract(Branch, [mixed(Alpha, JAlphaList)], Temporary),
  union([mixed(AlphaOne, JAlphaOneList),
    mixed(AlphaTwo, JAlphaTwoList)], Temporary, NewBranch),
  closes([NewBranch | Rest], D),
  combineJLists(Alpha, JAlphaOneList, JAlphaTwoList, JAlphaList).

/* BETA RULE */

closes([Branch | Rest], D) :-
  member(mixed(Beta,JBetaList), Branch),
  disjunctive(Beta),
  components(Beta, BetaOne, BetaTwo),
  subtract(Branch, [mixed(Beta, JBetaList)], Temporary),
  copy_term(Temporary, TempBranchOne),
  copy_term(Temporary, TempBranchTwo),
  union([mixed(BetaOne, _)], TempBranchOne, NewBranchOne),
  union([mixed(BetaTwo, _)], TempBranchTwo, NewBranchTwo),
  closes([NewBranchOne | Rest], D),
  closes([NewBranchTwo | Rest], D),
  combineBranches(TempBranchOne, TempBranchTwo, Temporary),
  member(mixed(BetaOne, JBetaOneList), NewBranchOne),
  member(mixed(BetaTwo, JBetaTwoList), NewBranchTwo),
  combineJLists(Beta, JBetaOneList, JBetaTwoList, JBetaList).

/* NU RULE */

/*  note that when a rule is applied to t(X), the signed formula is
    changed to u(X), marking the occurrence as used.
*/
```

```
closes([Branch | Rest], D) :-
  member(mixed(t(Var :: Fmla), List), Branch),
  subtract(Branch, [mixed(t(Var :: Fmla), List)], Temporary),
  union(Temporary, [mixed(u(Var :: Fmla), _), mixed(t(Fmla), NewList)],
    NewBranch),
  closes([NewBranch | Rest], D),
  deSharp(Var, NewList, List).

closes([], _D).

/*****************************************************************/

/* UTILITIES FOR TABLEAU CONSTRUCTION */

/*  The pi-rule in the tableau construction modifies an entire
    branch, using the so-called sharp operation.  Since mixed
    tableaus are now in use, sharp must take that into account.

    sharp(A, B) :- the S4 sharp operation applied to tableau branch A
    produces branch B.  This is designed to be applied to a
    mixed tableau, with the justification portion essentially
    ignored.  The S4 operation is not the same as the one for K.
    Also note that during application, occurrences of u(X) are
    changed back to t(X).
*/

sharp(X, Y) :- sharp_(X, Y), !.

sharp_([], []).

sharp_([mixed(u(V :: F), _) | Tail],
  [mixed(t(V :: F), _), mixed(t(F), _) | NewTail]) :-
    sharp_(Tail, NewTail).

sharp_([_Head|Tail], New) :-
  sharp_(Tail, New).


/*  trivial(List) :- Prolog variables are bound in List
    so that the result is a mixed tableau branch displaying
    a quasi-realization.
*/

trivial([mixed(_, Y) | Rest]) :- ground(Y), trivial(Rest).
trivial([mixed(t(X), [t(X)]) | Rest]) :- trivial(Rest).
trivial([mixed(f(X), [f(X)]) | Rest]) :- trivial(Rest).
trivial([mixed(u(X), [t(X)]) | Rest]) :- trivial(Rest).
trivial([]).
```

```
/*  negate(List, NegatedList) :-
    List is a list of signed formulas, and NegatedList is the result of
    negating each member.  That is, t(X) becomes t(neg X), and so on.
*/

negate([F | Rest], [G | NewRest]) :-
  unary(G),
  component(G, F),
  negate(Rest, NewRest).
negate([], []).

/*  combineBranches(InOne, InTwo, Out) :-
    Initially InOne, InTwo, and Out are lists (sets) that contain
    entries of the form mixed(Formula, JustificationList), with Formula
    instantiated in all three lists, but JustificationList
    instantiated in just the lists InOne and InTwo.  After execution,
    JustificationList in mixed(Formula, JustificationList),
    in the list Out is instantiated to be the union of the corresponding
    JustificationList parameters in InOne and InTwo, involving the
    same Formula.
*/

combineBranches([mixed(Formula, ListOne) | Rest], InTwo, Out) :-
    member(mixed(Formula, ListTwo), InTwo),
    union(ListOne, ListTwo, ListOut),
    member(mixed(Formula, ListOut), Out),
    combineBranches(Rest, InTwo, Out).

combineBranches([], _, _).

/*  combineJLists(Formula, List1, List2, CombinedList) :-
    List1 is a list of signed formulas having the same sign,
    List2 is a list of signed formulas having the same sign,
    CombinedList is the result of combining each member of List1
    with each member of List 2 to produce a formula of the same
    type as Formula, with the original members as components.
*/

combineJLists(Formula, [H|T], List2, CombinedList) :-
  combineSingle(Formula, H, List2, One),
  combineJLists(Formula, T, List2, Two),
  union(One, Two, CombinedList).

combineJLists(_, [], _, []).

/*  combineSingle(Formula, Comp1, List, NewList) :-
    NewList is the result of replacing each member Comp2
```

```
    of List with BinaryFormula, where
    constructBinary(Formula, Comp1, Comp2, BinaryFormula).
*/

combineSingle(Formula, Comp1, [H|T], NewList) :-
  combineSingle(Formula, Comp1, T, One),
  constructBinary(Formula, Comp1, H, Two),
  union(One, [Two], NewList).

combineSingle(_, _, [], []).

/*  constructBinary(Formula, Comp1, Comp2, BinaryFormula) :-
    all four variables are signed formulas, and BinaryFormula
    is constructed so that it has Comp1 and Comp2 as its
    components, and is of the same type as Formula---that is,
    both Formula and BinaryFormula are alpha or both are beta.
*/

constructBinary(Formula, AlphaOne, AlphaTwo, Alpha) :-
  conjunctive(Formula),
  conjunctive(Alpha),
  components(Alpha, AlphaOne, AlphaTwo).

constructBinary(Formula, BetaOne, BetaTwo, Beta) :-
  disjunctive(Formula),
  disjunctive(Beta),
  components(Beta, BetaOne, BetaTwo).

/*  deSharp(Var, List, NewList) :-
    Var is a justification variable. List is a list of justification
    formulas having sign t.  NewList is the result of replacing each
    member t(X) in List with t(Var :: X).
*/

deSharp(_, [],[]).
deSharp(Var, [t(X) | T], [t(Var :: X) | NewT]) :-
  deSharp(Var, T, NewT).

/*  disjunctionOfList(List, Disj) :-
    List consists of f-signed formulas, and Disj is
    the disjunction of these formulas.  This is used
    in the pi-rule.  Disjunction association is to the right.
*/

disjunctionOfList([f(A)], A).
disjunctionOfList([f(Head) | Tail], Head or X) :-
  disjunctionOfList(Tail, X).
```

```
/*************************************************************************/

/* BUILDING MIXED TABLEAUS AND DISPLAYING OUTPUT */

/*  test(X, D, PassOn) :-
    X is a modal formula and D is a non-negative number, representing
    modal depth.  X is annotated and a mixed tableau for the
    result, with an f sign, is constructed, using depth D.  If it
    succeeds in closing, success is anounced, the list of quasi-realizers
    is displayed, and values for the justification terms are shown.  PassOn
    instantiates to the list of quasi-realizers, so it can be passed on to the
    quasi-realizer to realizer conversion part.
*/

test(X, D, PassOn) :-
  reset,
  annotate(f(X), Y),
  closes([[mixed(Y, Q)]], D),
  nl, write('Propositional S4 tableau theorem'), nl, nl,
  write('Quasi-realizing list consists of:'), nl, nl,
  writeList(Q), nl,
  removeF(Q, PassOn),  /*this has been added*/
  write('where:'), nl, nl,
  writeQuasiReasons, nl.

writeList([f(Head) | Tail]) :- write(Head), nl, writeList(Tail).
writeList([]).

writeQuasiReasons :-
  proves(X, Y),
  getPremises(Y, Premises),
  getConclusions(Y, Conclusions),
  write(Premises), write(' -> '), write(X :: Conclusions), nl, fail.

writeQuasiReasons.

/*  getPremises(List, Premises) :-
    List is a branch of a mixed tableau.
    is the list of formulas of the form
    a :: b where t(a :: b) is on the
    justification list of a member of List.
*/

getPremises(List, Premises) :-
  getPremises_(List, Temp),
  removeT(Temp, Premises), !.

getPremises_([mixed(t(_ :: _), L) | Rest], Premises) :-
```

```
  getPremises_(Rest, Temp),
  union(Temp, L, Premises).

getPremises_([_ | Rest], Premises) :-
  getPremises_(Rest, Premises).

getPremises_([], []).

/*  removeT(List, NewList) :-
    List is a list of t-signed formulas and NewList is the
    result of stripping off the signs.
*/

removeT([t(X) | Rest], [X | NewRest]) :-
  removeT(Rest, NewRest).

removeT([], []).

/*  getConclusions(List, Conclusions) :-
    List is a branch of a mixed tableau.  Conclusions
    is the list of formulas f(F) where f(F) is on the
    justification list of a member of List.
*/

getConclusions(List, Conclusions) :-
  getConclusions_(List, Temp),
  removeF(Temp, Conclusions), !.

getConclusions_([mixed(f(_), L) | Rest], Conclusions) :-
  getConclusions_(Rest, Temp),
  union(Temp, L, Conclusions).

getConclusions_([_ | Rest], Conclusions) :-
  getConclusions_(Rest, Conclusions).

getConclusions_([], []).

/*  removeF(List, NewList) :-
    List is a list of f-signed formulas and NewList is the
    result of stripping off the signs.
*/

removeF([f(X) | Rest], [X | NewRest]) :-
  removeF(Rest, NewRest).

removeF([], []).

/*********************************************************/
```

```
/* NEXT THE CONVERSION FROM QUASI TO PROPER REALIZATION   */
/*********************************************************/

/* THE CONVERSION PROCESS */

/*  condense(QuasiSet, Signed, Realizer, Subst) :-
                        Signed
              QuasiSet --------> (Realizer, Subst)

    My notation is used above.  It represents the following.
    Signed is a signed modal formula.  QuasiSet is a set of
    quasi-realizers for Signed, and Realizer is a single
    realizer for it. Subst is a substitution, represented as a
    list of items, s(Var, Term), where Var is a variable and
    Term is a term.  If Signed has an f sign, the conjunction
    of QuasiSet, with Subst applied, should imply Realizer.
    If Signed has a t sign, Realizer should imply the
    disjunction of QuasiSet, with Subst applied.  This is
    what the algorithm guarantees.  The code below implements
    the steps of the algorithm, but does not verify that
    the implications hold.  QuasiSet and Signed are inputs,
    Realizer and Subst are outputs.  The propositional steps below
    are quite uniform, and could have been combined.
*/

/*Atomic case*/
condense([P], t(P), P, []) :- atom( P).
condense([P], f(P), P, []) :- atom(P).

/*T imp case*/
condense(QuasiSet, t(A imp B), Realizer, Subst) :-
  separateBinary(QuasiSet, QuasiA, QuasiB),
  condense(QuasiA, f(A), RealizerA, SubstA),
  condense(QuasiB, t(B), RealizerB, SubstB),
  sub(SubstB, RealizerA, Aprime),
  sub(SubstA, RealizerB, Bprime),
  Realizer = Aprime imp Bprime,
  union(SubstA, SubstB, Subst).

/*F imp case*/
condense(QuasiSet, f(A imp B), Realizer, Subst) :-
  separateBinary(QuasiSet, QuasiA, QuasiB),
  condense(QuasiA, t(A), RealizerA, SubstA),
  condense(QuasiB, f(B), RealizerB, SubstB),
  sub(SubstB, RealizerA, Aprime),
  sub(SubstA, RealizerB, Bprime),
  Realizer = Aprime imp Bprime,
  union(SubstA, SubstB, Subst).
```

```
/*T and case*/
condense(QuasiSet, t(A and B), Realizer, Subst) :-
  separateBinary(QuasiSet, QuasiA, QuasiB),
  condense(QuasiA, t(A), RealizerA, SubstA),
  condense(QuasiB, t(B), RealizerB, SubstB),
  sub(SubstB, RealizerA, Aprime),
  sub(SubstA, RealizerB, Bprime),
  Realizer = Aprime and Bprime,
  union(SubstA, SubstB, Subst).

/*F and case*/
condense(QuasiSet, f(A and B), Realizer, Subst) :-
  separateBinary(QuasiSet, QuasiA, QuasiB),
  condense(QuasiA, f(A), RealizerA, SubstA),
  condense(QuasiB, f(B), RealizerB, SubstB),
  sub(SubstB, RealizerA, Aprime),
  sub(SubstA, RealizerB, Bprime),
  Realizer = Aprime and Bprime,
  union(SubstA, SubstB, Subst).

/*T or case*/
condense(QuasiSet, t(A or B), Realizer, Subst) :-
  separateBinary(QuasiSet, QuasiA, QuasiB),
  condense(QuasiA, t(A), RealizerA, SubstA),
  condense(QuasiB, t(B), RealizerB, SubstB),
  sub(SubstB, RealizerA, Aprime),
  sub(SubstA, RealizerB, Bprime),
  Realizer = Aprime or Bprime,
  union(SubstA, SubstB, Subst).

/*F or case*/
condense(QuasiSet, f(A or B), Realizer, Subst) :-
  separateBinary(QuasiSet, QuasiA, QuasiB),
  condense(QuasiA, f(A), RealizerA, SubstA),
  condense(QuasiB, f(B), RealizerB, SubstB),
  sub(SubstB, RealizerA, Aprime),
  sub(SubstA, RealizerB, Bprime),
  Realizer = Aprime or Bprime,
  union(SubstA, SubstB, Subst).

/*T neg case*/

condense(QuasiSet, t(neg A), Realizer, Subst) :-
  separateNeg(QuasiSet, QuasiA),
  condense(QuasiA, f(A), Aprime, SubstA),
  Realizer = neg Aprime,
  Subst = SubstA.
```

```
/*F neg case*/

condense(QuasiSet, f(neg A), Realizer, Subst) :-
  separateNeg(QuasiSet, QuasiA),
  condense(QuasiA, t(A), Aprime, SubstA),
  Realizer = neg Aprime,
  Subst = SubstA.

/*T box case*/

condense(QuasiSet, t(box A), Realizer, Subst) :-
  separateBox(QuasiSet, QuasiA),
  condense(QuasiA, t(A), Aprime, SubstA),
  tBoxProcess(QuasiA, Aprime, SubstA, S),
  member(Var::_, QuasiSet),
  Sigma = [s(Var, S * Var)],
  sub(Sigma, Aprime, NewAprime),
  Realizer = Var::NewAprime,
  union(SubstA, Sigma, Subst).

/*F box case*/

condense(QuasiSet, f(box A), Realizer, Subst) :-
  combineBoxDisj(QuasiSet, QuasiA, f(A)),
  condense(QuasiA, f(A), Aprime, SubstA),
  fBoxProcess(QuasiSet, Aprime, SubstA, T),
  sub(SubstA, T::nothing, NewT::nothing),
  Realizer = NewT::Aprime,
  Subst = SubstA.


/*******************************************************/
/* SUBSTITUTION */

/*  sub(SubList, Formula, NewFormula) :-
    SubList is a list of members of the form s(Var, Term),
    where Var is a variable and Term is a justification term.
    Formula is a justification logic formula.
    NewFormula is the result of replacing all occurrences
    of Var in Formula with occurrences of Term, for every
    s(Var, Term) in SubList.  Important note: It is assumed
    that there is no "interference" between variables.  That
    is, replacing v by t does not introduce any variable that
    will, themselves, need replacing.  The "no new variable"
    condition in the algorithm guarantees this.  It greatly
    simplifies the programming.
*/
```

```
sub([], Formula, Formula).
sub([s(Var, Term) | Rest], Formula, NewFormula) :-
  subInFormula(Var, Formula, Term, Temp),
  sub(Rest, Temp, NewFormula).

/*  subInFormula(Var, Formula, Repl, NewFormula) :-
    the result of replacing all occurrences of variable Var
    in formula Formula with occurrences of Repl
    is NewFormula.  Does not check that Var is a variable.
*/

subInFormula(_Var, PropLetter, _Repl, PropLetter) :-
  atom(PropLetter).
subInFormula(Var, J :: A, Repl, NewJ :: NewA) :-
  subInTerm(Var, J, Repl, NewJ),
  subInFormula(Var, A, Repl, NewA).
subInFormula(Var, neg A, Repl, neg NewA) :-
  subInFormula(Var, A, Repl, NewA).
subInFormula(Var, A and B, Repl, NewA and NewB) :-
  subInFormula(Var, A, Repl, NewA),
  subInFormula(Var, B, Repl, NewB).
subInFormula(Var, A or B, Repl, NewA or NewB) :-
  subInFormula(Var, A, Repl, NewA),
  subInFormula(Var, B, Repl, NewB).
subInFormula(Var, A imp B, Repl, NewA imp NewB) :-
  subInFormula(Var, A, Repl, NewA),
  subInFormula(Var, B, Repl, NewB).

/*  subInTerm(Var, Term, Repl, NewTerm) :-
    the result of replacing all occurrences of variable Var
    in justification term Term with occurrences of Repl
    is NewTerm.  Does not check that Var is a variable.
*/

subInTerm(Var, Var, Repl, Repl).
subInTerm(Var, Atom, _Repl, Atom) :-
  atom(Atom), Atom \== Var.
subInTerm(Var, (One * Two), Repl, (NewOne * NewTwo)) :-
  subInTerm(Var, One, Repl, NewOne),
  subInTerm(Var, Two, Repl, NewTwo).
subInTerm(Var, (One + Two), Repl, (NewOne + NewTwo)) :-
  subInTerm(Var, One, Repl, NewOne),
  subInTerm(Var, Two, Repl, NewTwo).

/*********************************************************/
/* OTHER UTILITY METHODS USED BY CONVERSION PROCESS */
```

```
/*  separateBinary(List, First, Second) :-
    List is a list of binary formulas, all implications,
    or all conjunctions, or all disjunctions.  First becomes
    the list of left components, and Second the list
    of right components.  Union is used to avoid repetitions.
*/

separateBinary([], [], []).
separateBinary([Binary | Rest], ListOne, ListTwo) :-
  (Binary = A imp B; Binary = A and B; Binary = A or B),
  separateBinary(Rest, RestOne, RestTwo),
  union(RestOne, [A], ListOne),
  union(RestTwo, [B], ListTwo).

/*  separateNeg(List, NewList) :-
    List is a list of negated formulas.  NewList becomes
    the list these formulas with negations removed.
    Union is used to avoid repetitions.
*/

separateNeg([], []).
separateNeg([neg A | Rest], [A | NewRest]) :-
  separateNeg(Rest, NewRest).

/*  separateBox(List, NewList) :-
    List is a list of formulas of the form t::A.  NewList becomes
    the list these formulas with t removed.
    Union is used to avoid repetitions.
*/

separateBox([], []).
separateBox([_Term::A | Rest], [A | NewRest]) :-
  separateBox(Rest, NewRest).

/*  tBoxProcess(QuasiA, Aprime, SubstA, S) :-
    QuasiA, Aprime, and SubstA are inputs.  S is computed,
    where S = t1 +  + tk as in the written version of the
    algorithm.  Recorded for use later is information written
    as where(), consisting of a new justification term, and
    material from which the formula the term justifies can
    be constructed.  The formula is Aprime imp AiSubstA.
    A constant t is also recorded to distinguish information
    written by this case from that written by
    fBoxProcess below.
*/

tBoxProcess([Ai], Aprime, SubstA, S) :-
  gensym(t, Term),
```

```
  S = Term,
  assert(where(Term, Aprime, Ai, SubstA, t)).

tBoxProcess([Ai|Rest], Aprime, SubstA, S) :-
  gensym(t, Term),
  S = Term + RestS,
  tBoxProcess(Rest, Aprime, SubstA, RestS),
  assert(where(Term, Aprime, Ai, SubstA, t)).

/*  fBoxProcess(QuasiSet, Aprime, SubstA, T) :-
    QuasiSet, Aprime, and SubstA are inputs.  T is
    computed, where T = u1*t1 ++ uk*tk as in the
    written version of the algorithm.  Recorded for
    use later is information written as where(),
    consisting of a new justification term, and
    material from which the formula the term justifies can
    be constructed.  The formula is \/AiSubstA imp Aprime.
    A constant f is also recorded to distinguish information
    written by this case from that written by
    tBoxProcess above.
*/

fBoxProcess([Aterm::Adisj], Aprime, SubstA, T) :-
  gensym(t, Term),
  T = Term*Aterm,
  assert(where(Term, Aprime, Adisj, SubstA, f)).

fBoxProcess([Aterm::Adisj|Rest], Aprime, SubstA, T) :-
  gensym(t, Term),
  T = (Term*Aterm) + RestS,
  fBoxProcess(Rest, Aprime, SubstA, RestS),
  assert(where(Term, Aprime, Adisj, SubstA, f)).

/*  combineBoxDisj(InList, OutList, Type) :-
    boxed disjunctions of InList are combined into
    OutList, dropping the disjunctions and omitting
    the boxes.  Type is the kind of formula being
    disjuncted.  It uses disjunctToList, which see.
*/

combineBoxDisj([], [], _).

combineBoxDisj([_Term::Disj|Rest], List, Type) :-
  disjunctToList(Disj, Temp1, Type),
  combineBoxDisj(Rest, Temp2, Type),
  union(Temp1, Temp2, List).

/*  disjunctToList(Disjunct, List, Type) :-
```

```
    Disjunct is a disjunction.  List consists
    of the formulas that are disjuncted, where
    these are of type Type, taken to specify a
    quasi-realization set.  It is assumed that
    the sign involved is f.
*/

disjunctToList(D, L, T) :- disjunctToList_(D, L, T), !.

disjunctToList_(A or Rest, List, T) :-
  inQuasiSet(f(A), T),
  disjunctToList_(Rest, Temp, T),
  union(Temp, [A], List).

disjunctToList_(A, [A], _).

/*  inQuasiSet(F,G) :-
    signed justification formula F is a member
    of <<G>>, where G is a signed modal formula.
    Indexes are suppressed in G.
*/

inQuasiSet(A, A) :- propAtom(A).

inQuasiSet(AlphaQ, AlphaF) :-
  conjunctive(AlphaQ), conjunctive(AlphaF),
  components(AlphaQ, AlphaQone, AlphaQtwo),
  components(AlphaF, AlphaFone, AlphaFtwo),
  inQuasiSet(AlphaQone, AlphaFone),
  inQuasiSet(AlphaQtwo, AlphaFtwo).

inQuasiSet(BetaQ, BetaF) :-
  disjunctive(BetaQ), disjunctive(BetaF),
  components(BetaQ, BetaQone, BetaQtwo),
  components(BetaF, BetaFone, BetaFtwo),
  inQuasiSet(BetaQone, BetaFone),
  inQuasiSet(BetaQtwo, BetaFtwo).

inQuasiSet(UnaryQ, UnaryF) :-
  unary(UnaryQ), unary(UnaryF),
  component(UnaryQ, CQ), component(UnaryF, CF),
  inQuasiSet(CQ, CF).

inQuasiSet(t(_::Q), t(box F)) :-
  inQuasiSet(t(Q), t(F)).

inQuasiSet(f(_::Q), f(box F)) :-
  disjunctionFrom(f(Q), f(F)).
```

```
disjunctionFrom(A, F) :-
  disjunctionFrom_(A, F), !.

disjunctionFrom_(f(Q), f(F)) :-
  inQuasiSet(f(Q), f(F)).

disjunctionFrom_(f(A or B), f(F)) :-
  inQuasiSet(f(A), f(F)),
  disjunctionFrom_(f(B), f(F)).


/*************************************************************/

/* RUNNING AND DISPLAYING OUTPUT */

/*  convert(QuasiSet, ModalFormula) :-
    QuasiSet is a set of quasi-realizers, supplied as output
    of a quasi-realization program.  These are quasi-realizers
    of ModalFormula.  A realizer for ModalFormula is constructed
    from QuasiSet, displayed, and then additional information
    is displayed.
*/

convert(QuasiSet, ModalFormula) :-
  reset,
  condense(QuasiSet, f(ModalFormula), Realizer, _),
  write('A realization is:'), nl, nl,
  write(Realizer), nl, nl,
  write('where:'), nl, nl,
  writeReasons, nl.

writeReasons :-
  where(Term, Aprime, Ai, SubstA, t),
  sub(SubstA, Ai, AiSub),
  write(Term::(Aprime imp AiSub)), nl, fail.

writeReasons :-
  where(Term, Aprime, Adisj, SubstA, f),
  sub(SubstA, Adisj, AdisjSub),
  write(Term::(AdisjSub imp Aprime)), nl, fail.

writeReasons.

/*************************************************************/
/*              TOP LEVEL DRIVER                             */
/*************************************************************/
```

```
/*  realizer(ModalFormula) :-
    computes and displays information about a realizer for
    ModalFormula.  Quasi-realization information is also
    displayed.
*/

realizer(ModalFormula, ModalDepth) :-
  test(ModalFormula, ModalDepth, PassOn),
  convert(PassOn, ModalFormula).
```

5.2 Running the Program

The simplest way to describe running the program from Section 5.1 is to present examples. For purposes of Prolog input, the binary connectives are writen: `and`, `or`, and `imp`, in infix position. Negation is written in prefix position, as `neg`. (The more obvious `not` isn't used because it already has a meaning in some Prolog implementations.) The modal operator $\Box$ is written in prefix position, as `box`. Uppercase letters cannot be used as propositional variables, since Prolog would treat them as Prolog variables. We use `p`, `q`, and the like in our examples. There is also a propositional constant, `bot`, representing falsehood.

Example 5.2.1 Let us say we want a realizer for $(\Box P \vee \Box Q) \supset \Box(P \vee Q)$. The formula is supplied to Prolog as `(box p or box q) imp box(p or q)`. We also need to supply a *modal depth*. The purpose of this is discussed in Section 5.3—for this example we use 2. Then, the query entered is the following.

```
?- realizer((box p or box q) imp box(p or q), 2).
```

The response is the following printout.

```
Propositional S4 tableau theorem

Quasi-realizing list consists of:

((v1::p)or v2::q)imp j2::p or q
((v1::p)or v2::q)imp j1::p or q

where:

[v1::p] -> j1::[p or q]
[v2::q] -> j2::[p or q]

A realization is:

((v1::p)or v2::q)imp (t3*j1+t4*j2)::p or q

where:

t1::p imp p
```

```
t2::q imp q
t4:: (p or q)imp p or q
t3:: (p or q)imp p or q
```

Output begins with the announcement that the formula entered is, in fact, an S4 theorem. (If it were not, the query would simply fail.) It then gives the members of a set of quasi-realizers.

```
((v1::p)or v2::q)imp j2::p or q
((v1::p)or v2::q)imp j1::p or q
```

The usual notation $t{:}X$ cannot be used in output display since : already has a Prolog meaning. We have used :: in its place. Justification variables, in this example, are `v1, v2`. Justification terms introduced at the quasi-realization stage are `j1, j2`. Later, at the quasi-realization to realization conversion stage, `t3, t4` also appear. (Actually, `t1, t2` also turn up in the printout. These are used in intermediate calculations, and do not appear in the realization formula. They can be ignored.) Prolog omits unnecessary parentheses; `j2::p or q` should be read as `j2::(p or q)`.

What `[v1::p] -> j1::[p or q]` tells us is that justification term `j1` is chosen so that $v_1 : P \vdash_{\mathsf{LP}} j_1{:}(P \vee Q)$. In fact, $v_1{:}P \vdash_{\mathsf{LP}} P \vee Q$, and the Lifting Lemma 1.2.2 then ensures the existence of an appropriate justification term j_1. Similarly for j_2. All justification terms introduced at the quasi-realization stage are to be understood this way.

A different version of the Lifting Lemma is used for justification terms introduced at the conversion stage. This is the no-premise version, Corollary 1.2.3. Now `t3:: (p or q)imp p or q` tells us $\vdash_{\mathsf{LP}} t_3{:}[(P \vee Q) \supset (P \vee Q)]$, an easy internalization from $\vdash_{\mathsf{LP}} (P \vee Q) \supset (P \vee Q)$.

The most significant piece of output is the realizer itself, in the present case it is the following. `((v1::p)or v2::q)imp (t3*j1+t4*j2)::p or q`, or written in somewhat more conventional notation, $(v_1{:}P \vee v_2{:}Q) \supset (t_3 \cdot j_1 + t_4 \cdot j_2){:}(P \vee Q)$. The output can be simplified. This is discussed in Section 5.4.

Here is a second example. This time we just give the query and the printout. This is read similarly to Example 5.2.1.

Example 5.2.2 We ask for a realizer for $(\Box P \vee \Box Q) \supset \Box(\Box P \vee \Box Q)$. First, the query.

```
?- realizer((box p or box q) imp box(box p or box q),3).
```

And then the printout.

```
Propositional S4 tableau theorem

Quasi-realizing list consists of:

((v1::p)or v2::q)imp j4:: (v3::p)or j3::q
((v1::p)or v2::q)imp j2:: (j1::p)or v4::q

where:

[v1::p] -> j1::[p]
[v1::p] -> j2::[ (j1::p)or v4::q]
[v2::q] -> j3::[q]
[v2::q] -> j4::[ (v3::p)or j3::q]
```

```
A realization is:

((v1::p)or v2::q)imp (t7*j2+t8*j4):: ((t3*v3+t4*j1)::p)or (t5*j3+t6*v4)::q

where:

t1::p imp p
t2::q imp q
t4::p imp p
t3::p imp p
t6::q imp q
t5::q imp q
t8:: ((v3::p)or j3::q)imp ((t3*v3+t4*j1)::p)or (t5*j3+t6*v4)::q
t7:: ((j1::p)or v4::q)imp ((t3*v3+t4*j1)::p)or (t5*j3+t6*v4)::q
```

And a final example. Again without comment.

Example 5.2.3 We produce a realizer for $\Box((\Box p \vee \Box q) \supset \Box(p \vee q))$. The query.

```
?- realizer(box((box p or box q) imp box(p or q)), 3).
```

And the printout.

```
Propositional S4 tableau theorem

Quasi-realizing list consists of:

j3:: (((v1::p)or v2::q)imp j2::p or q)or ((v1::p)or v2::q)imp j1::p or q

where:

[v1::p] -> j1::[p or q]
[v2::q] -> j2::[p or q]
[] -> j3::[ ((v1::p)or v2::q)imp j2::p or q, ((v1::p)or v2::q)imp j1::p or q]

A realization is:

(t5*j3):: ((v1::p)or v2::q)imp (t3*j2+t4*j1)::p or q

where:

t1::p imp p
t2::q imp q
t4:: (p or q)imp p or q
t3:: (p or q)imp p or q
t5:: (((((t1*v1)::p)or (t2*v2)::q)imp j2::p or q)or
   (((t1*v1)::p)or (t2*v2)::q)imp j1::p or q)
   imp ((v1::p)or v2::q)imp (t3*j2+t4*j1)::p or q
```

The function of j_3 needs some comment. We have been a bit casual about reading quasi-realization 'where' conditions. The full form is `List1 -> t::List2`, and this should be understood as saying $\vdash_{\mathsf{LP}} C \supset t{:}D$ where C is the conjunction of members of `List1` and D is the disjunction of members of `List2`. In previous examples the lists involved were generally trivial. Note that conjunction of the empty list is taken to be true, and disjunction of the empty list is taken to be false, as usual.

Granted, t_5 is not an easy read either, but we leave that to you.

5.3 Modal Depth

For the logic K, tableaus give us an easy decision procedure. All tableau rules are single use, and every tableau rule reduces formula complexity. In particular, this is so for the passage from S to $S^\sharp$ in the rule $S, F\,\Box X \Rightarrow S^\sharp, F\,X$. Consequently every tableau construction (with single-use rule applications) must terminate. There are only a finite number of possible constructions so all can be tried systematically, yielding either a proof or the knowledge that there does not exist one.

For S4, things are not so simple. This logic uses a definition of $S^\sharp$ such that the passage from S to $S^\sharp$ does not necessarily reduce formula complexity. Then it is possible for a tableau construction to get stuck in a repeating loop. For instance, suppose we have a tableau branch consisting of $T\,\Box\neg\Box P, F\,\Box P$. Then we can proceed as follows.

1.	$T\,\Box\neg\Box P$
2.	$F\,\Box P$
1.	$T\,\Box\neg\Box P$
3.	$F\,P$
4.	$T\,\neg\Box P$
5.	$F\,\Box P$
1.	$T\,\Box\neg\Box P$
6.	$F\,P$
7.	$T\,\neg\Box P$
8.	$F\,\Box P$
	etc.

In each 'segment,' the presence of $F\,\Box P$ triggers a destructive rule application, but formula 1 always gets reproduced, so a fresh occurrence of $F\,\Box P$ can always reappear. The construction repeats, but does not terminate.

A decision procedure is still obtainable. Since all formulas appearing in a tableau are subformulas of the one being proved, only a finite number of formulas can appear. Then if a branch construction runs forever, some configuration must recur. When this happens, work on that branch can be terminated. Then, again, only a finite number of tableaus need be considered in a search for a proof.

Loop checking of this kind can be time consuming. What is often done is to set a *modal depth* bound for a proof attempt. One counts the number of destructive rule applications, $S, F\,\Box X \Rightarrow S^\sharp, F\,X$. When this number reaches a preassigned bound, one gives up. This does not provide a decision procedure. It does avoid looping. And any formula that has a proof will have one for some sufficiently high bound. It is also possible to calculate, in advance, what a sufficiently high number to take for a modal depth bound might be, but we have not done this.

In our queries, the second parameter is a modal depth bound. In Example 5.2.2, for instance, the query

```
?- realizer((box p or box q) imp box(box p or box q),3).
```

succeeds, but

```
?- realizer((box p or box q) imp box(box p or box q),2).
```

does not.

5.4 Simplifying the Output

Our Prolog implementation follows the algorithms blindly. This often leads to realizers that can be simplified considerably. Since $v : X \supset X$ is an LP axiom, an application of Lifting Lemma Corollary 1.2.3 tells us that there is a justification term t so that $v{:}X \supset t{:}X$. The usual proofs of this produce a term of some complexity. In fact, we can just take $t = v$. This often simplifies a quasi-realization list. For instance, in Example 5.2.2 we can take $j_1 = v_1$ and $j_3 = v_2$, and then the quasi-realizing list for this example becomes:

$$(v_1{:}P \vee v_2{:}Q) \supset j_4{:}(v_3{:}P \vee v_2{:}Q)$$
$$(v_1{:}P \vee v_2{:}Q) \supset j_2{:}(v_1{:}P \vee v_4{:}Q)$$

where

$$v_1{:}P \vdash_{\mathsf{LP}} j_2{:}(v_1{:}P \vee v_4{:}Q)$$
$$v_2{:}Q \vdash_{\mathsf{LP}} j_4{:}(v_3{:}P \vee v_2{:}Q).$$

At the stage where quasi-realizers are converted to realizers, the Lifting Lemma 1.2.2 itself was used. Now suppose we are told we have $t{:}(X \supset X)$. Then by Axiom A1 we conclude $v{:}X \supset t \cdot v{:}X$, and our algorithm makes use of this. But as above, $v{:}X \supset v{:}X$ will do as well for our purposes. In effect, it is as if t were an identity element.

Continuing with Example 5.2.2, all of t_1, t_2, t_3, t_4, t_5, and t_6 fall into the category just discussed. (Actually, t_1 and t_2 don't matter, since they don't occur in the computed realizer.) Then the computed provable realization reduces to the following

$$[v1{:}P \vee v_2{:}Q] \supset (t_7 \cdot j_2 + t_8 \cdot j_4){:}[(v_3 + v_1){:}P \vee (v_2 + v_4){:}Q]$$

where

$$\vdash_{\mathsf{LP}} t_8{:}[(v_3{:}P \vee v_2{:}Q) \supset ((v_3 + v_1){:}P \vee (v_2 + v_4){:}Q)]$$
$$\vdash_{\mathsf{LP}} t_7{:}[(v_1{:}P \vee v_4{:}Q) \supset ((v_3 + v_1){:}P \vee (v_2 + v_4){:}Q)]\,.$$

The reader might try applying simplifications to the easier Example 5.2.1.

5.5 Understanding the Program

The Prolog program in Section 5.1 is actually a combination of two earlier programs, one for computing quasi-realizers, the other for converting quasi-realizers to realizers. The two were written separately, then combined. The computation of quasi-realizers makes use of a simple S4 tableau theorem prover. It begins by constructing a closed tableau for $F\,X$ (assuming X is provable). Then it expands this to a mixed tableau and extracts a quasi-realizer from it. This quasi-realizer is passed on to the second part, which converts it into a realizer.

The program is divided into sections using lines of asterisks. Each section has a name, and we divide our discussion here into sections sharing those names.

5.5.1 /* SYNTAX */

The tableau construction uses signed formulas, written `t(Z)` and `f(Z)`. In Section 2.7 there is a brief discussion of single-usage for modal rules. The device used there was to cross an occurrence of $T\Box Z$ off after it had been used, and then remove the cross-off when a destructive rule has been applied. For this purpose the program uses a third sign, `u`, to represent a crossed off signed formula. Thus we have `t`, `f` and `u`.

Allowed propositional connectives are `neg`, `and`, `or`, and `imp`, with the obvious intended meanings. (`neg` was used instead of `not` because this already has a meaning in some Prolog implementations.) There is a propositional constant, `bot`, representing $\bot$ or falsehood. Also `box` is used for the necessitation operator. At the beginning of `/* SYNTAX */` there is a group of `op` codes. These establish that `neg` and `box` can be written in prefix position while the binary connectives can be written in infix position. They also specify that the binary connectives are right associative and have the same precedence. In addition `::` is introduced for :, right associative, and `*` for $\cdot$, left associative.

With this rather technical machinery out of the way, the program becomes much more readable. It is next specified which signed formulas act conjunctively, α cases, which disjunctively, β cases, and which are unary, signed negations. For each signed binary formula its components are defined, exactly as in the α and β tables of Table 2.1. Similarly for unary signed formulas.

Justification variables and terms have a special notation—variables are `vn` where `n` is an integer, terms are either `jn` or `tn`. Information about the usage of specific terms is written to the program workspace, database style, to be read back when output display is needed. `reset` clears away such writing from earlier program runs.

For use with the quasi-realization algorithm modal operators must be annotated, $\Box$ occurrences are replaced with indexed versions, $\Box_n$. Later on, negative occurrences of $\Box_n$ become justification variables, v_n. It simplifies things if we just replace all $\Box$ operators with justification variables at the start, but act as if they were indexed modal operators when appropriate. This is what `annotate` does.

5.5.2 /* TABLEAU CONSTRUCTION */

A mixed tableau is constructed by `closes`, which has two parameters. The second is a modal depth integer, whose purpose is discussed in Section 5.3. The first parameter is a mixed tableau, represented as a set of branches, which are sets of nodes. Nodes are of the form `mixed(modal, justification)`, where `modal` is a signed modal formula and `justification` is a set of signed justification formulas. A call on `closes` will succeed if the modal parts of the nodes constitute an `S4` tableau that can be continued to closure at the given modal depth. Corresponding values for the justification parts of the nodes are computed by the call.

As noted, a set representation is used for tableaus, where sets are represented as lists, order doesn't matter, and repetition is avoided. Modal formulas should have been rewritten previously so that they are annotated, and instead of $\Box_n X$ we have $v_n{:}X$. This is done by `annotate`, as described in Section 5.5.1.

The predicate `closes` makes strong use of Prolog's machinery. In effect, a closed tableau for an F-signed modal formula is first constructed, from the root node downward, then it is turned into a mixed tableau by filling in the justification formulas from branch ends upward.

Note that the Pi Rule case in the `closes` code makes use of `gensym` and `assert`. The first generates an atom, `jn` where `n` is an integer. This is to represent a particular justification term which is supposed to justify that the conjunction of the T-signed formulas on the tableau branch

implies the disjunction of the F-signed ones. Then `assert` writes a note to this effect, to the code base itself. This note will be read later on when it is time to write some output, in Section 5.5.4.

5.5.3 /* UTILITIES FOR TABLEAU CONSTRUCTION */

One of the tableau rules is $S, F\,\Box X \;\Rightarrow\; S^\sharp, F\,X$. The predicate `sharp` provides the operation converting a list of signed formulas to the corresponding sharped list, including restoring crossed out entries, signed with `u`, back to entries signed with `t`.

In Algorithm 3.4.1 two cases, Atomic and $F\,\Box$, call for a *trivial* expansion. A trivial expansion of signed formula M allows *any* S such that $S \subseteq \langle\!\langle M \rangle\!\rangle$. The predicate `trivial` produces a particularly simple trivial expansion. It makes use of the fact that we have chosen to represent $\Box_1 P$, for instance, by $v_1{:}P$.

In expanding a tableau construction to a mixed tableau, Algorithm 3.4.1 calls for negating members of a list, in the Negation Cases, and combining lists using binary connectives, in the α and β Cases. The first is handled by `negate`, and the second by `constructJLists`. In the same algorithm the β case combines two branches, $\mathcal{B}^{E_1}$ and $\mathcal{B}^{E_2}$ using a kind of pointwise union. This is taken care of by `combineBranches`. In the $T\,\Box$ case of the algorithm, if $T\,Z$ is in the set S_1 then $T\,v_k{:}Z$ is added to set S. This is a kind of inverse to the sharp operation, and `deSharp` is used for the purpose. Finally, in the $F\,\Box$ case of the algorithm, the set S is used to create a disjunction, $\bigvee S$, something that is the job of `disjunctionOfList`.

5.5.4 /* BUILDING MIXED TABLEAU AND DISPLAYING OUTPUT */

It was noted earlier that the Prolog program of Section 5.1 is the result of combining a quasi-realization program with a conversion from quasi-realization to realization program. The part of the code now being discussed is the end of the quasi-realization part.

The predicate `test` takes three parameters. If we were just interested in computing quasi-realizers, the first two would be enough. These are the modal formula we want a quasi-realizer for, and a modal depth parameter. The third parameter is instantiated to be the list of quasi-realizers, to be passed on to the second part of the program which converts quasi-realizers to realizers. In execution, `test` begins by clearing away results from earlier program runs, annotates a signed version of the formula, and calls on `closes`. If this call succeeds, it announces that fact, displays the list of quasi-realizers using `writeList`, and displays the role each justification term should play, using `writeQuasiReasons`. These displayed roles are extracted from the program itself, thought of as a database to which the information has been added during execution of `closes`.

5.5.5 /* THE CONVERSION PROCESS */

In Definition 4.4.1 we introduced notations $\mathcal{A} \xrightarrow{T\,A} (A', \sigma)$ and $\mathcal{A} \xrightarrow{F\,A} (A', \sigma)$. These correspond exactly to `condense(QuasiSet, Signed, Realizer, Subst)` where `QuasiSet` represents $\mathcal{A}$, `Signed` represents $T\,A$ or $F\,A$, `Realizer` represents A', and `Subst` represents σ. Consult Section 4.4 for further discussion. The code for `condense` directly implements Algorithm 4.4. Beyond the appearance of signed formulas seen above the arrows here, signed formulas play little role in this part of the program.

5.5.6 /* SUBSTITUTION */

Substitutions are represented as lists whose members are of the form `s(Var, Term)`, where `Var` is a justification variable and `Term` is a term to be substituted for it. `sub(Sublist, Formula,`

`NewFormula)` succeeds if `NewFormula` is the result of carrying out the substitutions in `Sublist` on the justification formula `Formula`. Since the substitutions that come up during the course of our algorithm don't interfere with each other, multiple substitutions can be carried out in any order, and we make use of this to simplify the code for `sub`. Non-interference means that domains are not shared, and output of one substitution does not involve a variable in the domain of another substitution.

5.5.7 /* OTHER UTILITY METHODS USED BY CONVERSION PROCESS */

This section is a collection of utility methods used in the code for `condense`.

The predicate `separateBinary` splits a list of binary formulas having the same connective into a list of first components and a list of second components. For instance, the list `[a imp b, c imp d, e imp f]` would separate into the list `[a, c, e]` and the list `[b, d, f]`. `separateNeg` does something analogous with negations, and `separateBox` with necessitated formulas.

In the $T\Box$ case of Algorithm 4.4.3 a justification term s is computed as $t_1 + \ldots + t_k$ for particular t_i. In the program this is effected by `tBoxProcess(QuasiA, Aprime, SubstA, S)`. New justification terms of the form `tn` are introduced and assigned to formulas `Aprime imp AiSubstA`, where `Aprime` is a parameter of `tBoxProcess`, `Ai` is a member of the list `QuasiA`, and `SubstA` is understood as having been applied to `Ai`. `S` is instantiated to be the summation of the justification terms introduced. Also, information recording all this is written to the program, thought of as a database, using `assert`.

The predicate `fBoxProcess` does an analogous thing to `tBoxProcess`, appropriate to the $F\Box$ case of Algorithm 4.4.3. There is one peculiar complication involved in the $F\Box$ case, however. Disjunctions must be separated into components, but those components must be members of $\langle\!\langle Z \rangle\!\rangle$, where Z is the signed formula over the arrow in the algorithm. Then a disjunction $A \vee B$ might separate into $[A, B]$, or into $[A \vee B]$, depending on the structure of Z. This is the job of `disjunctToList`, which calls on `inQuasiSet`, a predicate that amounts to an implementation of Definition 3.2.1.

Finally `combineBoxDisj(InList, OutList, Type)` is used for the $F\Box$ case of Algorithm 4.4.3. `InList` is a list of necessitated disjunctions of formulas of type `Type`, and `OutList` is a list of all the formulas involved, with necessitation operators and disjunctions omitted. In terms of the Algorithm statement, if `InList` is $\{t_1{:}\bigvee \mathcal{A}_1, \ldots, t_k{:}\bigvee \mathcal{A}_k\}$ then `OutList` should be $\mathcal{A}_1 \cup \ldots \cup \mathcal{A}_k$. The remaining clauses in this section of the program define relations that are used by `combineBoxDisj`; in particular, this is where `disjunctToList` is used.

5.5.8 /* RUNNING AND DISPLAYING OUTPUT */

As noted earlier, our Prolog program was the result of combining two separate programs, one to compute quasi-realizers, the other to convert a set of quasi-realizers to a realizer. This section of code is the end of what was the second of these programs. `convert(QuasiSet, ModalFormula)` succeeds if `QuasiSet` is a set of quasi-realizers for `ModalFormula`. Along the way, its call on `condense` computes a realizer and records information about it. The realizer is displayed, and then the recorded information is displayed appropriately, by `writeReasons`.

5.5.9 /* TOP LEVEL DRIVER */

This final section consists of the simple code for `realizer`. It takes two parameters, the first a modal formula, the second a modal depth. It calls on `test`, in effect running the first of the two programs that were merged into the combined program. Then it calls on `convert`, running the second.

Chapter 6

What Next?

Justification logics began with LP, relating it to S4. It was quickly realized that a number of well-known modal logics have justification counterparts, and the list is slowly expanding. The methods presented here have wider applicability than the single case discussed, but if this case is understood, many extensions are straightforward. We have written Prolog code for realizing the modal logic K in the justification logic J, for instance. So, what is the likely range of possible extensions?

Destructive tableau systems exist for a number of standard modal logics, but not for those involving symmetry. Work on destructive tableaus goes back to [9] and before, and has had much subsequent development. The implementation given here traces back, in part, to [10]. We have built S4 and K quasi-realizers using this machinery, and there seems to be no barrier to extending the work to the whole range of modal logics having destructive tableau proof systems.

A broader range of modal logics have prefixed tableau systems, or equivalently, nested sequent systems, [13]. Realization proofs have been given for such systems in a uniform way [16]. It may be that quasi-realization algorithms could be developed making use of prefixed or nested sequent systems. If so, it is likely to be a simpler thing than full realization. Implementation of prefixed tableau proof systems is not difficult, so this is a promising direction to persue.

The quasi-realization to realization conversion algorithm that we gave really does not depend on the particular logic involved. What is needed is that the logic have a lifting lemma, and $+$ and $\cdot$ should be part of the language. A stand-alone Prolog implementation for conversion exists. This could be paired with a suitable quasi-realization algorithm arising from prefixed tableau/nested sequent systems.

Much depends on the Lifting Lemma. The proof of that is constructive, but as far as I know there are no good implementations of it. This could use some serious thinking.

Recently justification logics have been extended to admit quantifiers, [3, 8]. The assumption is that first-order modal logics are *monotonic*, and the Barcan formula is not a validity. Then FOLP, first-order LP, has been shown to be realizable in first-order S4, [3]. In fact, the algorithms presented here extend to admit quantifiers. Quasi-realization to realization can be found in [14], but the proof that there are quasi-realizers for FOLP given there is non-constructive. A constructive algorithm exists, but has not been published.

First-order justification logics are new territory. A range of familiar first-order monotonic modal logics are known to have justification counterparts. The status of constant domain logics is unknown, and ripe for investigation.

What started out as part of an elegant and enlightening solution to a problem involving the status of intuitionistic logic has turned out to be a more general phenomenon. Its extent is unclear. There is something new under the sun, but so far only some of it is actually in daylight.

Bibliography

[1] S. N. Artemov. Explicit provability and constructive semantics. *Bulletin of Symbolic Logic*, 7(1):1–36, Mar. 2001.

[2] S. N. Artemov. The logic of justification. *The Review of Symbolic Logic*, 1(4):477–513, Dec. 2008.

[3] S. N. Artemov and T. Yavorskaya (Sidon). First-order logic of proofs. Technical Report TR–2011005, CUNY Ph.D. Program in Computer Science, May 2011.

[4] S. Feferman, editor. *Kurt Gödel Collected Works*. Oxford, 1986-2003. Five volumes.

[5] M. Fitting. The logic of proofs, semantically. *Annals of Pure and Applied Logic*, 132(1):1–25, Feb. 2005.

[6] M. Fitting. Realizations and LP. *Annals of Pure and Applied Logic*, 161(3):368–387, Dec. 2009. Published online August 2009.

[7] M. Fitting. Justification logics and hybrid logics. *Journal of Applied Logic*, 8(4):356–370, Dec. 2010. Published online August 2010.

[8] M. Fitting. Possible world semantics for first order LP. Technical Report TR–2011010, CUNY Ph.D. Program in Computer Science, Sept. 2011.

[9] M. C. Fitting. *Proof Methods for Modal and Intuitionistic Logics*. D. Reidel Publishing Co., Dordrecht, 1983.

[10] M. C. Fitting. First-order modal tableaux. *Journal of Automated Reasoning*, 4:191–213, 1988.

[11] M. C. Fitting. The logic of proofs, semantically. *Annals of Pure and Applied Logic*, 132:1–25, 2005.

[12] M. C. Fitting. Reasoning with justifications. In D. Makinson, J. Malinowski, and H. Wansing, editors, *Towards Mathematical Philosophy*, number 28 in Trends in Logic, chapter 6, pages 107 – 123. Springer, 2009.

[13] M. C. Fitting. Prefixed tableaus and nested sequents. *Annals of Pure and Applied Logic*, 163:291–313, 2012. Available on-line at http://dx.doi.org/10.1016/j.apal.2011.09.004.

[14] M. C. Fitting. Realization using the Model Existence Theorem. Submitted, 2013.

[15] K. Gödel. Eine Interpretation des intuistionistischen Aussagenkalkuls. *Ergebnisse eines mathematischen Kolloquiums*, 4:39–40, 1933. Translated as *An interpretation of the intuitionistic propositional calculus* in [4] I, 296-301.

[16] R. Goetschi and R. Kuznets. Realization for justification logics via nested sequents: Modularity through embedding. *Annals of Pure and Applied Logic*, 163(9):1271–1298, Sept. 2012. Published online March 2012.

[17] R. M. Smullyan. A unifying principle in quantification theory. *Proceedings of the National Academy of Sciences*, 49(6):828–832, June 1963.

[18] R. M. Smullyan. *First-Order Logic*. Springer-Verlag, Berlin, 1968. Revised Edition, Dover Press, New York, 1994.