Using Zippers for Nested Sequents with Focus

MSc Thesis *(Afstudeerscriptie)*

written by

Shing Yau Simon Chiu

under the supervision of **Dr Marianna Girlando** and **Dr Malvin Gattinger**, and submitted to the Examinations Board in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam*.

Date of the public defense:	**Members of the Thesis Committee:**
August 25, 2025	Dr Benno van den Berg (chair)
	Dr Marianna Girlando
	Dr Malvin Gattinger
	Dr Ronald de Haan

Institute for Logic, Language and Computation

Abstract

Nested sequent calculus augments the formalism of standard Gentzen-style sequents by allowing nesting of sequents, giving them a tree-like structure. In the classical modal setting, the resulting calculus is invertible, enabling terminating root-first proof search without backtracking. While provers based on nested sequents exist, they involve additional structure and heuristics not formally part of the proof system, indicating a gap between theory and implementation. Here, we augment nested sequents by adding a current focus, internalising the heuristics of proof search into the calculus. We implement this using the zipper data structure in Haskell, and test it on various examples to ensure correctness.

Contents

Chapter 1

Introduction

1.1 Background

Sequent calculus was introduced by Gentzen as a tool for formal analysis of proofs. It is used to construct analytic proofs, where every formula in the proof is a subformula of the conclusion. This property makes the sequent calculus particularly suited for backwards proof search, as it limits the formulas that could occur in a proof.

An example of a sequent calculus is the system **G3cp** for classical propositional logic, which can be found in [Nv01]. The rules of **G3cp** are all invertible, meaning that no information is lost when applying a rule. In the context of proof search, this means that the order of application of rules does not matter, which streamlines the proof search process.

Basic modal logic extends classical modal logic with the modal operators $\Box$ and $\Diamond$. There are a number of sequent calculi for modal logic which extend classical propositional logic by adding additional rules, such as the system **G3K** for modal logic K, obtained by adding to **G3cp** the following rule:

$$\frac{\Gamma \Rightarrow A}{\Gamma', \Box\Gamma \Rightarrow \Box A, \Delta}\,\mathsf{K}$$

However, this rule is not invertible, as information is lost in backwards application of K when choosing which formulas to include in the context Γ' and Δ. Consequently, this means that backwards proof search for **G3K** requires backtracking, for a failed proof search may be due to a wrong choice of context in the application of the K rule.

Sequent calculus has been generalised in various ways by adding additional structure. One such extension is nested sequent calculus, introduced independently in [Bul92], [Kas94], [Brü09], and [Pog09], which allows nesting of sequents on the right of the arrow $\Rightarrow$. This gives nested sequents a tree-like structure, mirroring the tree model property of many modal logics. The resulting calculus is fully invertible, which enables proof search without backtracking.

Theorem provers are computer programs which automate the proof search procedure, often based on existing proof systems. While many proof search algorithms already exist on paper, practical design choices have to be made in the representation of the proof system and implementation of the algorithm, which may involve adding additional structure or heuristics.

1.2 Contribution

This thesis aims to bridge the gap between theory and implementation of nested sequents for modal logic K. We introduce the system NS^f_K, which uses *nested sequents with focus*, extending the nested sequent formalism of [Brü09] by adding a focus, the current location within the nested sequent. We then implement a prover based on NS^f_K in Haskell. To represent nested sequents with focus, we make use of *zippers*, a technique for representing hierarchal data structures such as lists and trees, introduced in [Hue97]. The source code is available on `https://github.com/shosukeyuu/NestProveML`.

Chapter 2 reviews the basics of modal logic, including the syntax, semantics, and axiomatisation, as well as the nested sequent proof system. Chapter 3 introduces the zipper data structure, which is illustrated with lists and trees in Haskell. Chapter 4 presents nested sequents with focus and the proof system NS^f_K, along with the proof search algorithm. Chapter 5 shows the implementation of NS^f_K in Haskell, and we test the implementation using example formulas.

1.3 Previous Work

This thesis is inspired by [Yan24], which implements a prover in Haskell based on **G3K** using the zipper data structure to represent proof trees to facilitate backtracking. We instead use zippers to represent nested sequents, as no backtracking is needed. The MOLTAP prover presented in [van09] is based on nested sequents with signed formulas, using state monads to facilitate proof search. By introducing nested sequents with focus, we provide a more streamlined proof search strategy. The MOIN prover presented in [GS20] implements a prover for both classical and intuitionistic modal logics based on nested sequents in Prolog. Utilising types and data structures in Haskell, our implementation represents and manipulates the tree-like structures of proofs and nested sequents more concisely.

Chapter 2

Modal Logics and Proof

In this chapter, we will cover the basics of modal logic, as well as nested sequent calculi as introduced in [Brü09], which uses one-sided sequents. Standard two-sided sequents have the form $\Gamma \Rightarrow \Delta$, where Γ and Δ are multisets of formulas, interpreted as $\bigwedge \Gamma \implies \bigvee \Delta$. Classically, this is equivalent to $\bigvee\{\neg\gamma \mid \gamma \in \Gamma\} \vee \bigvee \Delta$, therefore every sequent is equivalent to a sequent where the antecedent is empty. The advantage of a one-sided approach is that it allows for a uniform presentation of rules, instead of having separate left and right rules, which also makes it simpler to implement. The tradeoff is that we will have to work with the modal language in negation normal form, requiring both $\wedge$, $\vee$, and $\Box$, $\Diamond$ to be primitive.

2.1 Syntax

Definition 2.1. For a countable set of propositional variables Prop, the language $\mathcal{L}$ of modal logic in negation normal form is generated by the following:

$$A ::= p \mid \bar{p} \mid A \vee A \mid A \wedge A \mid \Diamond A \mid \Box A \ ,$$

where $p \in \mathsf{Prop}$.

We use $A, B, C, ...$ to denote formulas, and $p, q, r, ...$ to denote propositional variables.

Definition 2.2. The negation $\overline{A}$ is extended to any arbitrary formula A by the following rewrite rules:

$$\begin{aligned} \overline{\bar{p}} &:= p \\ \overline{A \vee B} &:= \overline{A} \wedge \overline{B} \\ \overline{A \wedge B} &:= \overline{A} \vee \overline{B} \\ \overline{\Diamond A} &:= \Box \overline{A} \\ \overline{\Box A} &:= \Diamond \overline{A} \ . \end{aligned}$$

We define implication $A \to B$ as $\overline{A} \vee B$, and $\top$ and $\bot$ as $p \vee \bar{p}$ and $p \wedge \bar{p}$ respectively, for some fixed p.

Definition 2.3. For any formula A and propositional variable p, $A[\frac{B}{p}]$ is obtained from A by replacing each occurrence of p by B.

2.2 Semantics

Definition 2.4. A Kripke frame $\mathcal{F}$ is a pair (W, R), where

- W is a non-empty set of worlds.
- $R \subseteq W \times W$ is a binary relation on W.

We call $R[w] := \{u \in W \mid w\ R\ u\}$ the set of successors of w.

Definition 2.5. A Kripke model $\mathcal{M}$ is a pair $(\mathcal{F}, V)$, where

- $\mathcal{F} = (W, R)$ is a Kripke frame,
- $V : \mathsf{Prop} \to \mathcal{P}(W)$ is a valuation.

We will refer to Kripke frames and Kripke models as just frames and models respectively.

Definition 2.6. Given a model $\mathcal{M} = (W, R)$, world $w \in W$, and formula A, the satisfaction or forcing relation $\mathcal{M}, w \vDash A$ is defined inductively as follows:

$$
\begin{array}{lll}
\mathcal{M}, w \vDash p & \text{iff} & w \in V(p), \text{ for } p \in \mathsf{Prop}; \\
\mathcal{M}, w \vDash \bar{p} & \text{iff} & w \notin V(p), \text{ for } p \in \mathsf{Prop}; \\
\mathcal{M}, w \vDash A \vee B & \text{iff} & \mathcal{M}, w \vDash A \text{ or } \mathcal{M}, w \vDash B; \\
\mathcal{M}, w \vDash A \wedge B & \text{iff} & \mathcal{M}, w \vDash A \text{ and } \mathcal{M}, w \vDash B; \\
\mathcal{M}, w \vDash \Diamond A & \text{iff} & \mathcal{M}, v \vDash A, \text{ for some } v \in R[w]; \\
\mathcal{M}, w \vDash \Box A & \text{iff} & \mathcal{M}, v \vDash A \text{ for all } v \in R[w].
\end{array}
$$

We call a model-world pair $(\mathcal{M}, w)$ a *pointed model*.

Definition 2.7. For any formula A:

- A is *satisfiable* if there are $\mathcal{M}, w$ such that $\mathcal{M}, w \vDash A$.
- A is *valid* if $\mathcal{M}, w \vDash A$, for any pointed model $\mathcal{M}, w$.

We denote a valid formula A by $\vDash A$.

2.3 Axioms

In this section, we introduce the standard Hilbert-style axiomatisation of modal logic. Hilbert-style systems consist of axioms and rules. A *proof* is a finite sequence of formulas, each being either an axiom, or following from previous items in the sequence by a rule. A formula is *provable* if it is the last formula of some proof. If $\mathcal{S}$ is a system, we write $\vdash_{\mathcal{S}} A$ to denote that A is provable in $\mathcal{S}$.

Definition 2.8. The Hilbert system K consists of the following axioms:

- all classical propositional tautologies;
- $\Box(p \to q) \to (\Box p \to \Box q)$;
- $\Diamond p \leftrightarrow \neg\Box\neg p$

The rules of inference of K are the following:

- Modus ponens: If $\vdash_{\mathsf{K}} A$ and $\vdash_{\mathsf{K}} A \to B$, then $\vdash_{\mathsf{K}} B$;
- Uniform substitution: If $\vdash_{\mathsf{K}} A$, then for any $p \in \mathsf{Prop}$ and formula B, $\vdash_{\mathsf{K}} A[\frac{B}{p}]$;
- Necessitation: If $\vdash_{\mathsf{K}} A$, then $\vdash_{\mathsf{K}} \Box A$.

Two properties of a good proof system are *soundness* and *completeness*. A system is sound if only validities are provable, and it is complete if all validities are provable. Together they imply that the proof system fully reflects all validities of the logic.

Theorem 2.9 (Soundness)**.** If $\vdash_{\mathsf{K}} A$, then $\vDash A$.

Soundness is proven by showing that all axioms are valid, and that all rules preserves validity.

Theorem 2.10 (Completeness)**.** If $\vDash A$, then $\vDash_{\mathsf{K}} A$.

The proof for normal modal logics can be found in [BRV10, Theorem 4.22].

2.4 Nested Sequents

This section introduces nested sequent calculi, presented in [Brü09].

Definition 2.11. A nested sequent Γ is a finite multiset of formulas and boxed sequents. A boxed sequent is an expression $[\Gamma]$, where Γ is a nested sequent.

From here onwards, we will use *sequents* to refer to nested sequents. Sequents are denoted by Γ, Δ, and have the following form:

$$\Gamma = A_1, ..., A_m, [\Delta_1], ..., [\Delta_n] \ ,$$

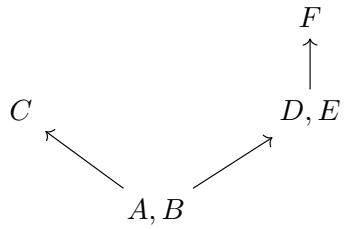
where $n, m \geq 0$.

Definition 2.12. The *corresponding formula* $\mathcal{I}(\Gamma)$ of a sequent $\Gamma = A_1, ..., A_m, [\Delta_0], ..., [\Delta_n]$ is defined as

$$\mathcal{I}(\Gamma) := A_1 \vee ... \vee A_m \vee \Box\mathcal{I}(\Delta_1) \vee ... \vee \Box\mathcal{I}(\Delta_n) \ .$$

Every sequent can be represented as tree, where the nodes are multisets of formulas. The root of the tree is the multiset $A_1, ..., A_m$, and the immediate subtrees are the trees corresponding to the sequents $\Delta_1, ..., \Delta_n$.

For example, consider the sequent $A, B, [C], [D, E[F]]$.



Due to the tree-like structure of sequents, the rules of the calculus that we will present can be applied not just to the formulas at the root of a sequent, but anywhere deep in the structure. In order to state the rules, a notational device is needed. A context $\Gamma\{\}$ is a sequent with a hole, which can be filled in by another seqeunt. For example, if $\Gamma\{\} = A, [B, \{\}]$, then $\Gamma\{C, [D]\} = A, [B, C, [D]]$.

Definition 2.13. A hole is the expression $\{\}$. The set of contexts is defined inductively as follows:

1. the singleton multiset $\{\}$ is a context;
2. if Δ is a sequent and $\Gamma\{\}$ is a context, then $\Delta, \Gamma\{\}$ is a context;
3. if $\Gamma\{\}$ is a context, then $[\Gamma\{\}]$ is a context.

A sequent system consists of rules, which have the following form:

$$\frac{\Gamma_1, ..., \Gamma_n}{\Delta}\,\rho$$

We call $\Gamma_1, ..., \Gamma_n$ the premises, and Δ the conclusion. Axioms are rules with no premise. A *derivation* of a sequent Γ is a finite upward-growing tree with root Γ whose nodes are sequents, and are built according to the rules of inference. We call the sequents at the leaves of a derivation the *premises*, and the sequent at the root the *conclusion*. A *closed derivation*, or *proof*, is a derivation whose premises are all axioms. Similarly to Hilbert systems, we say a sequent Γ is *derivable* if there is a proof of Γ in $\mathcal{S}$, which is denoted by $\vdash_{\mathcal{S}} \Gamma$.

Definition 2.14. The sequent calculus $\mathsf{NS_K}$ consists of the following rules:

$$\frac{}{\Gamma\{p, \bar{p}\}}\,\mathsf{ax} \qquad \frac{\Gamma\{A\} \quad \Gamma\{B\}}{\Gamma\{A \wedge B\}}\,\wedge \qquad \frac{\Gamma\{A, B\}}{\Gamma\{A \vee B\}}\,\vee$$

$$\frac{\Gamma\{[A]\}}{\Gamma\{\Box A\}}\,\Box \qquad \frac{\Gamma\{\Diamond A, [\Delta, A]\}}{\Gamma\{\Diamond A, [\Delta]\}}\,\Diamond$$

Figure 2.1: Rules of $\mathsf{NS_K}$

Here is an example of a proof of the formula $\Diamond(p \vee q) \to \Diamond p \vee \Diamond q$, which is $\Box(\bar{p} \wedge \bar{q}) \vee \Diamond p \vee \Diamond q$

in negation normal form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\bar{p},p],\Diamond p,\Diamond q}\,\mathsf{ax}}{[\bar{p}],\Diamond p,\Diamond q}\,\Diamond \quad \dfrac{\dfrac{}{[\bar{q},q],\Diamond p,\Diamond q}\,\mathsf{ax}}{[\bar{q}],\Diamond p,\Diamond q}\,\Diamond}{[\bar{p}\wedge\bar{q}],\Diamond p,\Diamond q}\,\wedge}{\Box(\bar{p}\wedge\bar{q}),\Diamond p,\Diamond q}\,\Box}{\Box(\bar{p}\wedge\bar{q}),\Diamond p\vee\Diamond q}\,\vee}{\Box(\bar{p}\wedge\bar{q})\vee\Diamond p\vee\Diamond q}\,\vee$$

A rule ρ is *admissible* in a system $\mathcal{S}$, if whenever the premises of ρ are provable, so is the conclusion; It is *invertible* if whenever the conclusion is provable, so are the premises.

The calculus $\mathsf{NS_K}$ admits the usual structural rules such as weakening and contraction. The admissibility of the necessitation rule is needed to prove soundness.

$$\dfrac{\Gamma}{[\Gamma]}\,\mathsf{nec} \qquad \dfrac{\Gamma\{\varnothing\}}{\Gamma\{\Delta\}}\,\mathsf{wk} \qquad \dfrac{\Gamma\{\Delta,\Delta\}}{\Gamma\{\Delta\}}\,\mathsf{ctr}$$

Figure 2.2: necessitation, weakening, and contraction

Lemma 2.15. The rules $\mathsf{nec}, \mathsf{wk}, \mathsf{ctr}$ are admissible in $\mathsf{NS_K}$.

What distinguishes the nested sequent formalism compared to other sequent systems for K is that all the rules are invertible. This implies that when constructing a proof, rules can be applied in any order to any applicable principal formula.

Lemma 2.16. All rules of $\mathsf{NS_K}$ are invertible.

The system $\mathsf{NS_K}$ is both sound and complete with respect to Kripke semantics.

Theorem 2.17 (Soundness)**.** If $\vdash_{\mathsf{NS_K}} \Gamma$, then $\vDash \mathcal{I}(\Gamma)$.

Theorem 2.18 (Completeness)**.** If $\vDash \mathcal{I}(\Gamma)$, then $\vdash_{\mathsf{NS_K}} \Gamma$.

Full proofs can be found in [Brü09]. Similar to the Hilbert system K, soundness is proven by showing that the initial sequent $\Gamma\{p,\bar{p}\}$ is valid, and that all rules preserves validity. Completeness is proven by extracting a countermodel from a failed exhaustive proof search of a related system. The proof search procedure for $\mathsf{NS_K}$ is explained in Section 4.1. A cut-elimination procedure is also provided in [Brü09], which offers another way to prove completeness.

Chapter 3

Zippers

Zippers are data structures used to navigate and modify hierarchal structures, introduced in [Hue97]. This is done by splitting the data structure into two parts: the *focus*, and the *context*. The *focus* is the current position which can be modified, and the *context* or *path* points to the location where the focus is within the structure. In the following, we will illustrate two main examples of zippers: Lists and trees.

3.1 Lists

In Haskell, lists are ordered sequences of elements of a type. For example, `[5, 4, 8, 2, 0, 6]` is a list of integers. For any arbitrary type `a`, the list type for `a` is denoted by `[a]`. The goal of a zipper for lists is to store a current position within the list, allowing efficient definition of navigation and modification functions such as changing the value, deleting it, and moving left or right. We define a type class `ListLike` for the functions we would want from a zipper for lists:

```
class ListLike l where
-- create a zipped list from a single item
  singleton :: a -> l a
-- change the value at the current position
  change :: a -> l a -> l a
-- delete the current item and move to the right
  delete :: l a -> l a
-- move to the left
  moveLeft :: l a -> l a
-- move to the right
  moveRight :: l a -> l a
-- insert an item after the current item
  insertAfter :: a -> l a -> l a
```

We define a zipper for lists as follows:

```
data ZipList a = Zip [a] a [a]
```

As an example, the zipped list of `[5, 4, 8, 2, 0, 6]` focused on the fourth item `2` would be `Zip [8, 4, 5] 2 [0, 6]`. Note that the first list is reversed, since Haskell has access to the first items of a list before the latter, this makes it more efficient to, for example, move the focus to the left.

We define a `ListLike` instance for `ZipList`:

```
instance ListLike ZipList where
  singleton c = Zip [] c []

  change c (Zip xs _ ys) = Zip xs c ys

  delete (Zip xs _ (y:ys)) = Zip xs y ys
  delete (Zip _ _ [])      = error "Cannot delete last element"

  moveLeft (Zip (x:xs) c ys) = Zip xs x (c:ys)
  moveLeft (Zip [] _ _ )     = error "Cannot move left at first item"

  moveRight (Zip xs c (y:ys)) = Zip (c:xs) y ys
  moveRight (Zip _ _ [])      = error "Cannot move right at last item"

  insertAfter p (Zip xs c ys) = Zip xs c (p:ys)
```

3.2 Trees

Trees are a more complex example of a hierarchal data structure. In Haskell, a standard definition of the tree type of an arbitrary type `a` is:

```
data Tree a = Node a [Tree a]
```

That is, a tree is a node containing a value of type `a`, along with a list of its immediate subtrees. For example, the tree in Figure 3.1 would be be represented by the following:

```
example32 :: Tree Int
example32 = Node 1 [Node 2 [], Node 3 [Node 5 [], Node 6 []], Node 4 []]
```

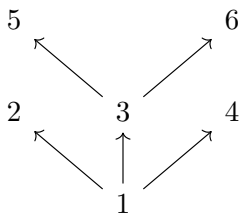


Figure 3.1: example tree

Similarly to lists, a zipper for trees stores a current location within the tree. We define the type class `TreeLike` for functions used to navigate within a tree:

```
class TreeLike z where
-- create a zipped tree with one node
  zsingleton :: a -> z a
-- move to the sibling on the left
  move_left :: Eq a => z a -> z a
-- move the the sibling on the right
  move_right :: Eq a => z a -> z a
-- move to the parent
  move_parent :: z a-> z a
-- move to the left-most child
  move_child :: Eq a => z a -> z a
```

The zipper for trees is defined as follows:

```
data ZipTree a = ZT (Tree a) (Path a)
data Path a = Top | Step a (Path a) [Tree a] [Tree a]
```

A zipper for a tree focuses on a subtree, with the path being the location of the root of that subtree within the whole tree. If the path of the focused subtree is `Top`, it means that the focus is in fact the whole tree. Otherwise, it is one `Step` from it's parent with value in `a`, who itself has a path, along with a list of siblings on both the left and right, which are subtrees. As a `ZipTree` has immediate access to its parent, children, and close siblings, it allows for efficient navigation and modification of values.

The zipper for the tree 3.1, focused on the subtree whose root is 3 is represented as follows:

```
example322 :: ZipTree Int
example322 = ZT (Node 3 [Node 5 [], Node 6 []]) (Step 1 Top [Node 2 []] [Node 4 []])
```

We define an instance of `TreeLike` for `ZipTree`:

```
instance TreeLike ZipTree where
  zsingleton x = ZT (Node x []) Top

  move_left (ZT t (Step s p (x:xs) ys)) = ZT x (Step s p xs (t:ys))
  move_left _                           = error "cannot move to left"

  move_right (ZT t (Step s p xs (y:ys))) = ZT y (Step s p (t:xs) ys)
  move_right _                            = error "cannot move to right"

  move_parent (ZT t (Step s p xs ys)) = ZT (Node s (xs ++ t:ys)) p
  move_parent _                       = error "cannot move to parent"

  move_child (ZT (Node c (t:ts)) p) = ZT t (Step c p [] ts)
  move_child _                      = error "cannot move to child"
```

We will use zippers in Section 5.2 to represent nested sequents with focus, which will be introduced in Section 4.2.

Chapter 4

Nested Sequents with Focus

4.1 Proof search in NS_K

The introduction of a *focus* for nested sequents is motivated by the implementation of proof search for NS_K. The following is the proof search algorithm for NS_K, adapted from [Brü09], which uses a cumulative version of the rules of NS_K, where the principal formula is repeated in each of the premises, and a rule is not applied to a sequent if no new formulas are added. This will be explained in the next section, when we introduce NS^f_K, which also uses cumulative rules.

Algorithm 1 Proof Search for NS_K

```
repeat
    while any non-□ rule ρ is applicable do
        apply ρ
    if □ rule is applicable then
        apply □
until no rules are applicable
```

Note that in the context of proof search, a rule is "applied" from the conclusion to the premises. Given a sequent, at each iteration of proof search, the principal formula of applicable rules could a priori be anywhere deep in the sequent. Therefore, the entire sequent has to be scanned through to look for applicable rules.

For example, consider the sequent $\Diamond A, [A, \Box E, [C \vee D]]$

$$\begin{array}{c} C \vee D \\ \uparrow \\ A, \Box E \\ \uparrow \\ \Diamond A \end{array}$$

This sequent has two possible rule applications: a $\Box$ rule and a $\vee$ rule. We do not apply the $\Diamond$ rule with principal formula $\Diamond A$, since its child already contains A. Since the non-$\Box$ rules have a higher priority, starting from the root we have to scan the whole tree before finding the formula $C \vee D$. Once we apply the $\vee$ rule, we have to scan through the entire tree again to make sure there are no non-$\Box$ rules to apply before applying the $\Box$ rule.

However, we can avoid the situation above if we start our proof search with a formula and not a sequent. Observe that a rule application during the while loop does not create new nodes, since only the $\Box$ rule creates a new boxed sequent. Only when the other rules are exhausted do we apply the $\Box$ rule to create a new child node. The only possible rule application then to the parent node would be if there is a $\Diamond$ formula present for the $\Diamond$ rule, or other $\Box$ formulas. Since no rule moves a formula from a child to its parent, these are the only possible further rule applications to the original parent node.

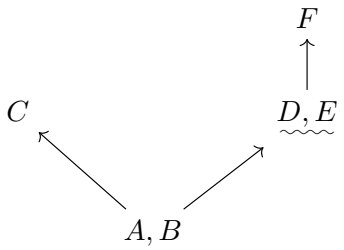
Therefore, we can avoid the need to scan the whole tree and instead focus on only one node at any given time: Starting with a single formula at the root, apply all propositional rules before creating a child node with the $\Box$ rule. Then we shift our focus to the newly created node, applying all propositional rules and potentially the $\Diamond$ rule, bringing in a formula if a $\Diamond$ formula is present in the parent node. This process is repeated, applying rules and moving the focus to the newly created child node after an application of $\Box$ rule, until no rules are applicable. Then, we shift our focus back along the branch until we find a $\Box$ formula to apply the $\Box$ rule, and the search continues, terminating either when we have an initial sequent, or the focus returns back to the root with no further applicable rules.

The rest of this chapter introduces *nested sequents with focus*, which internalise the heuristics described above into the calculi.

4.2 Nested Sequents with Focus

Definition 4.1. A *nested sequent with focus* is a nested sequent with a distinguished subsequent, which is either the sequent itself or a boxed sequent, called the *focus*, denoted by a wavy underline $\underset{\sim}{\Gamma}$.

We call nested sequents with focus *focused sequents* for short. Consider the previous example of a sequent $A, B, [C], [D, E, [F]]$. The same sequent with focus on the boxed sequent $[D, E, [F]]$ is denoted by $A, B, [C], [\underset{\sim}{D, E, [F]}]$, and the corresponding tree is as follows:



For simplicity, we only underline the root of our focus in the tree representation, and we may identify the focus of a sequent with the formulas in its root, as there is a one-to-one correspondence between nodes of a tree and its subtrees.

Before presenting the rules, we introduce the notion of set sequents, which will be used to limit rule application to stop proof search.

Definition 4.2. The *set sequent* of the sequent

$$A_1, ..., A_m, [\Delta_1], ..., [\Delta_n]$$

is the underlying set of

$$A_1, ..., A_m, [\Lambda_1], ..., [\Lambda_n] \ ,$$

where $\Lambda_1, ..., \Lambda_n$ are the set sequents of $\Delta_1, ..., \Delta_n$ respectively.

For example, the set sequent of $p, p, [p, q, q, [r]]$ is $p, [p, q, [r]]$.

Definition 4.3. The rules of $\mathsf{NS}^f_{\mathsf{K}}$ are as follows:

$$\frac{}{\Gamma\{\underset{\sim}{p}, \underset{\sim}{\bar{p}}\}}\,\mathsf{ax}^f \qquad \frac{\Gamma\{\underset{\sim}{A \wedge B}, \underset{\sim}{A}\} \quad \Gamma\{\underset{\sim}{A \wedge B}, \underset{\sim}{B}\}}{\Gamma\{\underset{\sim}{A \wedge B}\}}\,\wedge^f \qquad \frac{\Gamma\{\underset{\sim}{A \vee B}, \underset{\sim}{A}, \underset{\sim}{B}\}}{\Gamma\{\underset{\sim}{A \vee B}\}}\,\vee^f$$

$$\frac{\Gamma\{\Box A, \underset{\sim}{[A]}\}}{\Gamma\{\underset{\sim}{\Box A}\}}\,\Box^f \qquad \frac{\Gamma\{\Diamond A, \underset{\sim}{[\Delta, A]}\}}{\Gamma\{\Diamond A, \underset{\sim}{[\Delta]}\}}\,\Diamond^f, \qquad \frac{\Gamma\{\underset{\sim}{\Delta}, \underset{\sim}{[\Lambda]}\}}{\Gamma\{\Delta, \underset{\sim}{[\Lambda]}\}}\,\mathsf{bk}$$

The rules come with the following conditions:

1. for all rules, the set sequents of all its premises are different from the set sequent of the conclusion;
2. for the $\Box^f$ rule, the node of the active formula $\Box A$ in the conclusion does not have a child node containing A.

The structural rule bk stands for *back*, as we move "back" to the parent when reading the rule from the conclusion to the premise.

Note that $\Gamma\{\underset{\sim}{\Delta}\}$ means that Δ is part of the focus of the sequent. All the rules of $\mathsf{NS}^f_{\mathsf{K}}$ are *cumulative*, meaning that the active formula in the conclusion is repeated in all the premises. For the $\Diamond^f$ rule, which is also cumulative in the original NS_{K}, it ensures that contraction is admissible. For the other rules, it acts as a history of formulas we have encountered before during proof search to ensure termination.

The following is an example of the proof of the K-axiom $\Box(p \to q) \to (\Box p \to \Box q)$, which reduces to $\Diamond(p \wedge \bar{q}) \vee \Diamond\bar{p} \vee \Box q$ in negation normal form. The principal formulas in the premises

of non-$\Diamond$ rules have been omitted for readability:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Diamond(p \wedge \bar{q}), \Diamond\bar{p}, [\underset{\sim}{q}, p, \bar{p}]}\,\mathsf{ax}^f}{\Diamond(p \wedge \bar{q}), \Diamond\bar{p}, [\underset{\sim}{q}, p]}\,\Diamond^f \quad \dfrac{}{\Diamond(p \wedge \bar{q}), \Diamond\bar{p}, [\underset{\sim}{q}, \bar{q}]}\,\mathsf{ax}^f}{\Diamond(p \wedge \bar{q}), \Diamond\bar{p}, [\underset{\sim}{q}, p \wedge \bar{q}]}\,\wedge^f}{\Diamond(p \wedge \bar{q}), \Diamond\bar{p}, [\underset{\sim}{q}]}\,\Diamond^f}{\Diamond(p \wedge \bar{q}), \Diamond\bar{p}, \Box q}\,\Box^f}{\Diamond(p \wedge \bar{q}), \Diamond\bar{p} \vee \Box q}\,\vee^f}{\Diamond(p \wedge \bar{q}) \vee \Diamond\bar{p} \vee \Box q}\,\vee^f$$

4.3 Proving in $\mathsf{NS}^f_{\mathsf{K}}$

By design, the rules of $\mathsf{NS}^f_{\mathsf{K}}$ impose an order of application of rules. The only rules which change the focus are $\Box^f$, which simultaneously creates a new boxed sequent and move the focus to it, and bk, which moves the focus one step back to the parent. This means that once bk is used to move out of a boxed sequent, there is no way to return the focus back to it.

Therefore, the proof search strategy is to always apply all non-$\Box^f$ logical rules, including the ax^f rule, whenever possible. Once those are exhausted, we apply the $\Box^f$ rule to create a new boxed sequent and move the focus to it, and the process is repeated, until we reach a boxed sequent where no $\Box^f$ application is possible and is not an initial sequent. Then, the bk rule is applied to move back along the branch. Since the non-$\Box^f$ rules have already been exhausted in the parent sequent, the only possible application would be a $\Box^f$ if there are multiple $\Box$ formulas present in the sequent, or the bk rule to search for an unused $\Box$ formula along the branch. This terminates either when we reach an initial sequent, or the bk rule returns to the root and no rules are applicable.

Algorithm 2 Proof search for $\mathsf{NS}^f_{\mathsf{K}}$

Require: Formula A
Initialise: $\underset{\sim}{A}$
repeat
  if any non-$\Box$, non-bk rule ρ^f is applicable **then**
    apply ρ^f
  else if $\Box^f$ rule is applicable **then**
    apply $\Box^f$
  else if current focus is not the root **then**
    apply bk rule
until no rules are applicable

Note the use of the ax^f rule is included in the first **if** conditional, which also terminates the proof search since the rule has no premise.

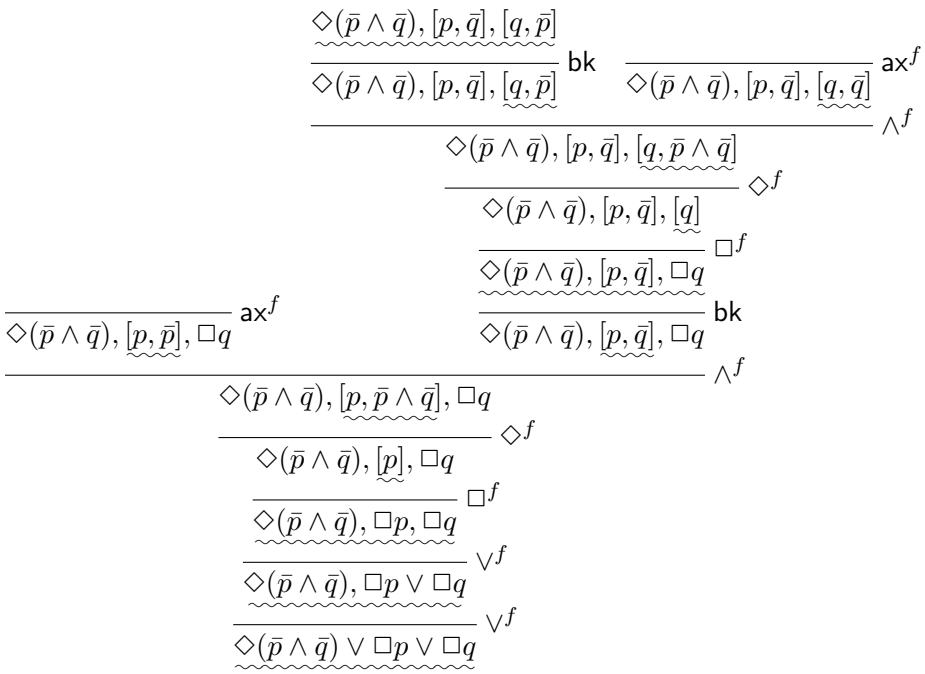


Figure 4.1: failed proof of $\Diamond(\bar{p} \wedge \bar{q}) \vee \Box p \vee \Box q$

Figure 4.1 is an example of a failed proof search of the invalid formula $\Box(p \vee q) \to \Box p \vee \Box q$, which is $\Diamond(\bar{p} \wedge \bar{q}) \vee \Box p \vee \Box q$ in negation normal form. Principal formulas in the premises of non-$\Diamond^f$ rules have been omitted for clarity.

The systems $\mathsf{NS_K}^f$ and $\mathsf{NS_K}$ are equivalent in the following way:

Theorem 4.4. For any formula A,

$$\vdash_{\mathsf{NS}_\mathsf{K}^f} \underset{\sim}{A} \qquad \Longleftrightarrow \qquad \vdash_{\mathsf{NS_K}} A$$

Proof. ($\Longrightarrow$) Suppose we have a proof of $\underset{\sim}{A}$ in NS_K^f. By the first side condition of the rules, each non-$\Diamond$ formula can be the principal formula of at most one rule application. Therefore, we can remove repeated instances of principal formulas of all non-$\Diamond^f$ rule applications, and ignore the focus and bk rule to construct a $\mathsf{NS_K}$ proof of A.

($\Longleftarrow$) Suppose we have a proof of A in $\mathsf{NS_K}$. We first add principal formulas for all rule applications to the premises to make the rules cumulative. If there are redundant applications of the $\Diamond$ rule, in which a $\Diamond$ formula is applied to the same boxed sequent multiple times, we ignore such applications, which is illustrated in Figure 4.2. By invertibility of $\mathsf{NS_K}$, we permute the rule applications in a way which respects the focus. New rule applications may need to be added, but by invertibility, no information is lost, so this does not affect other rule applications. Finally, we add the focus to the sequents to obtain a proof in NS_K^f. □

Combining this result with Theorem 2.17 and Theorem 2.18, we immediately obtain soundness and completeness of NS_K^f.

Corollary 4.5. For any formula A,

- (Soundness) If $\vdash_{\mathsf{NS}_\mathsf{K}^f} \underset{\sim}{A}$, then $\vDash A$;

$$\dfrac{\dfrac{\dfrac{\dfrac{\Diamond(A \vee B), [C, A, B, A \vee B]}{\Diamond(A \vee B), [C, A, B]}\Diamond}{\Diamond(A \vee B), [C, A \vee B]}\vee}{\Diamond(A \vee B), [C]}\Diamond}{} \quad \rightsquigarrow \quad \dfrac{\dfrac{\dfrac{\Diamond(A \vee B), [C, A, B]}{\Diamond(A \vee B), [C, A \vee B]}\vee}{\Diamond(A \vee B), [C]}\Diamond}{}$$

Figure 4.2: redundant application of $\Diamond$ rule

- (Completeness) If $\vDash A$, then $\vdash_{\mathsf{NS}^f_{\mathsf{K}}} \underset{\sim}{A}$.

Chapter 5

Haskell Implementation

In this chapter, we present a prover for NS^f_K based on Algorithm 2 in Haskell. The full code can be found on `https://github.com/shosukeyuu/NestProveML`.

5.1 Syntax

We implement the formulas of modal logic in negation normal form as `MNNForm`, indexing our set of propositional variables with `Int`:

```
type Proposition = Int
data MNNForm = P Proposition
  | NP Proposition
  | Con MNNForm MNNForm
  | Dis MNNForm MNNForm
  | Box MNNForm
  | Dia MNNForm
  deriving (Eq,Ord,Show)
```

Other operators are defined in terms of the primitive ones, and are used for easier input of formulas:

```
neg :: MNNForm -> MNNForm
neg (P n) = NP n
neg (NP n) = P n
neg (Con f g) = Dis (neg f) (neg g)
neg (Dis f g) = Con (neg f) (neg g)
neg (Box f) = Dia (neg f)
neg (Dia f) = Box (neg f)

implies, iff :: MNNForm -> MNNForm -> MNNForm
implies f = Dis (neg f)
iff f g = Con (implies f g) (implies g f)

top, bot :: MNNForm
top = Dis (P 0) (NP 0)
bot = Con (P 0) (NP 0)

bigDis :: [MNNForm] -> MNNForm
bigDis [x]    = x
bigDis (x:xs) = Dis x (bigDis xs)
bigDis []     = bot
```

For example, the formula $\Diamond p_0 \wedge \Box(\bar{p}_0 \vee p_1)$ is represented as follows:

```
example5_1 :: MNNForm
example5_1 = Con (Dia (P 0)) (Box (Dis (NP 0) (P 1)))
```

To make it more readable, we define a pretty printing function `pp`, which converts an `MNNForm` into a `String`. We do this by defining a type class `PrettyPrint`, so that we can use the same function name `pp` to pretty print other types, such as sequents:

```
class PrettyPrint a where
  pp :: a -> String

instance PrettyPrint MNNForm where
  pp :: MNNForm -> String
  pp (P n) = "P" ++ show n
  pp (NP n) = "~P" ++ show n
  pp (Con f g) = "(" ++ pp f ++ " & " ++ pp g ++ ")"
  pp (Dis f g) = "(" ++ pp f ++ " v " ++ pp g ++ ")"
  pp (Box f) = "{}" ++ pp f
  pp (Dia f) = "<>" ++ pp f
```

Using the previous example, we have that

```
ghci> pp example5_1
"(<>P0 & {}(~P0 v P1))"
```

We also define an `Arbitrary` instance for `MNNForm` which generates arbitrary formulas, used for testing the correctness of the prover in Section 5.5.

```
instance Arbitrary MNNForm where
  arbitrary = sized randomForm where
    randomForm 0 = oneof [P <$> choose (1,7), NP <$> choose (1,7)]
    randomForm n = oneof [ Con <$> randomForm (n `div` 7) <*> randomForm (n `div` 7)
                         , Dis <$> randomForm (n `div` 7) <*> randomForm (n `div` 7)
                         , Box <$> randomForm (n `div` 7)
                         , Dia <$> randomForm (n `div` 7)]
```

5.2 Sequents

Recall that a nested sequent is a multiset of formulas and boxed sequents. We define the type of nested sequents as follows, where `f` is a formula type, which will be `MNNForm` in our case:

```
newtype NestSeq f = NS [Either f (NestSeq f)]
  deriving (Eq, Ord, Show)
```

We choose the `List` type to represent sequents, since they are the simplest option, and they more closely resemble the syntax of sequents. While lists are ordered, it does not affect provability of formulas. Because lists can only contain terms of a fixed type, we use the `Either` type to tag formulas with `Left` and boxed sequents with `Right`. For example, the sequent $p_1, [\bar{p}_1, p_1 \vee p_2, [\Box p_3]]$ can be represented by the following:

```
example521 :: NestSeq MNNForm
example521 = NS [Left p, Right (NS [Left (neg p), Left (Dis p q), Right (NS [Left (Box
     r)])])] where
  p = P 1
  q = P 2
  r = P 3
```

As the constructors `Left`, `Right`, and `NS` make it difficult to read, we define an instance of `PrettyPrint` for `NestSeq` as follows:

```
instance PrettyPrint f => PrettyPrint (NestSeq f) where
  pp :: NestSeq f -> String
  pp (NS []) = ""
  pp (NS [Left x])      = pp x
  pp (NS (Left x:xs))   = pp x ++ ", " ++ pp (NS xs)
  pp (NS [Right y])     = "[" ++ pp y ++ "]"
  pp (NS (Right y:xs)) = "[" ++ pp y ++ "]" ++ ", " ++ pp (NS xs)
```

We then have:

```
ghci> pp example521
"P1, [~P1, (P1 v P2), [{}P3]]"
```

We define functions `forms` and `boxedSeqs` to extract the formulas and boxed sequents of a nested sequent as follows:

```
forms :: NestSeq f -> [f]
forms (NS sq) = lefts sq

boxedSeqs :: NestSeq f -> [NestSeq f]
boxedSeqs (NS sq) = rights sq
```

These are used to define the corresponding formula of a nested sequent:

```
correspondingForm :: NestSeq MNNForm -> MNNForm
correspondingForm ns = case (forms ns, boxedSeqs ns) of
  ([],  [])   -> bot
  (_:_, [])   -> bigDis $ forms ns
  ([],  _:_)  -> bigDis $ map (Box . correspondingForm) (boxedSeqs ns)
  (_:_, _:_)  -> Dis (bigDis $ forms ns) (bigDis $ map (Box . correspondingForm) (
      boxedSeqs ns))
```

For example,

```
ghci> pp (correspondingForm example521)
"(P1 v {}((~P1 v (P1 v P2)) v {}{}P3))"
```

To represent focused sequents, we use a zipper similar to that of basic trees in Section 3.2:

```
data PathNestSeq f = TopNS | StepNS [Either f (NestSeq f)] [Either f (NestSeq f)] (
    PathNestSeq f)
  deriving(Eq, Ord, Show)

data ZipNestSeq f = ZNS {focus :: [Either f (NestSeq f)], path :: PathNestSeq f }
  deriving(Eq, Ord, Show)
```

A `PathNestSeq` of `TopNS` means that the focus is the root. The first and second arguments of `StepNS` are the sequents on the left and right of the focus respectively, followed by the path of the parent sequent. The focus of a `ZipNestSeq` is the sequent corresponding to the subtree with whose root is the focus of the focused sequent we are representing. Note that we use the type `[Either f (NestSeq f)]` instead of `NestSeq f` for the arguments, since this removes the `NS` constructor in front, which makes it more straightforward to define functions.

We define an instance of `PrettyPrint` for `ZipNestSeq`, which surrounds the focus with `*`:

```
instance (Show f, PrettyPrint f) => PrettyPrint (ZipNestSeq f) where
  pp (ZNS sq pa) = case ppPathNestSeq pa of
    ([],[])   -> "*" ++ pp (NS sq) ++ "*"
```

```
    ([],r:rs) -> "*[" ++ pp (NS sq) ++ "]*, " ++ r:rs
    (l:ls,[]) -> l:ls ++ ", *[" ++ pp (NS sq) ++ "]*"
    (l:ls,r:rs) -> l:ls ++ ", *[" ++ pp (NS sq) ++ "]*, " ++ r:rs


ppPathNestSeq :: (Show f, PrettyPrint f) => PathNestSeq f -> (String,String)
ppPathNestSeq TopNS = ("", "")
ppPathNestSeq (StepNS ls rs pa) = case ppPathNestSeq pa of
  ([],[])   -> (pp (NS ls), pp (NS rs))
  ([],y:ys) -> ('[': pp (NS ls), pp (NS rs) ++ "], " ++ y:ys)
  (x:xs,[]) -> (x:xs ++ ", [" ++ pp (NS ls), pp (NS rs) ++ "]")
  (x:xs,y:ys) -> (x:xs ++ ", [" ++ pp (NS ls), pp (NS rs) ++ "], " ++ y:ys)
```

For example, consider the focused sequent $p_0, [p_0, [p_2, [p_1]], [p_2]], p_1$, which is represented as follows:

```
example522 :: ZipNestSeq MNNForm
example522 = ZNS [Left r, Right (NS [Left q])] (StepNS [Left p] [Right (NS [Left r])]
    (StepNS [Left p] [Left q] TopNS)) where
  p = P 0
  q = P 1
  r = P 2
```

Pretty printing it gives the following:

```
ghci> pp example522
"P0, [P0, *[P2, [P1]]*, [P2]], P1"
```

We define helper functions `toZipNS` and `fromZipNS` to convert between `NestSeq` and `ZipNestSeq`:

```
toZipNS :: NestSeq f -> ZipNestSeq f
toZipNS (NS sq) = ZNS sq TopNS

fromZipNS :: ZipNestSeq f -> NestSeq f
fromZipNS (ZNS cs TopNS) = NS cs
fromZipNS (ZNS cs (StepNS ls rs pa)) = (fromZipNS . move_parent) (ZNS cs (StepNS ls rs
     pa))
```

We also define an instance of `TreeLike` for `ZipNestSeq` to navigate through the tree-like structure of a focused sequent:

```
instance TreeLike ZipNestSeq where
  zsingleton x = ZNS [Left x] TopNS

  move_left (ZNS cs (StepNS ls rs pa)) =
    case rights ls of
      NS x :_ -> ZNS x (StepNS (delete (Right $ NS x) ls) (Right (NS cs):rs) pa)
      []      -> error "cannot go left"
  move_left (ZNS _ TopNS) = error "cannot go left"

  move_right (ZNS cs (StepNS ls rs pa)) =
    case rights rs of
      NS x :_ -> ZNS x (StepNS (Right (NS cs):ls) (delete (Right $ NS x) rs) pa)
      []      -> error "cannot go right"
  move_right (ZNS _ TopNS) = error "cannot go right"

  move_parent (ZNS cs (StepNS ls rs pa)) = ZNS (ls ++ Right (NS cs):rs) pa
  move_parent (ZNS _ TopNS) = error "no parent to move to"

  move_child (ZNS cs pa) =
    case rights cs of
```

```
      NS x:_ -> ZNS x (StepNS (take (fromJust $ elemIndex (Right $ NS x) cs) cs) (drop
           (1 + fromJust (elemIndex (Right $ NS x) cs)) cs) pa)
      []     -> error "no child to move to"
```

5.3 Prover

The data structures for rules and proofs are adapted from [Yan24]. We take `RuleName` to be strings. The `Proof` type is a modified tree type, where a node is either `Proved`, indicating that the branch is closed, or has the form `Node (ZipNestSeq f) RuleName [Proof f]`, where the first argument is the sequent of that node, followed by the `RuleName` of the rule with the first argument as its conclusion, and a list of proofs for the premises of the rule.

```
type RuleName = String

data Proof f = Proved | Node (ZipNestSeq f) RuleName [Proof f]
  deriving (Eq ,Ord ,Show)
```

A `Proof` is closed when all its leaves are `Proved`:

```
isClosedPf :: Eq f => Proof f -> Bool
isClosedPf Proved = True
isClosedPf (Node _ _ ts) = ts /= [] && all isClosedPf ts
```

We now define our rule type. Given any sequent type `s`, a `Rule s` takes a sequent as an input, and outputs a list of pairs for each possible application of the rule, where a rule is applied from the conclusion to the premise, and each pair consists of the `RuleName` and a list of sequents as its premises.

```
type Rule s = s ->[(RuleName, [s])]
```

Observe that Algorithm 2 groups rules into two categories: rules which do not create new boxed sequents and rules which do. To allow for extension of this implementation to other logics such as T and S4, we let the `Logic` be an input to our prover, and call the former `simpleRules` and latter `boxRules`

```
data Logic f = Log {simpleRules :: Rule f , boxRules :: Rule f}
```

The logical rules for $\mathsf{NS}_{\mathsf{K}}^{f}$ are implemented as follows, packaged into a logic `k`:

```
simpleAnd :: Rule (NestSeq MNNForm)
simpleAnd (NS sq) = [("&", [NS (Left f : sq), NS (Left g : sq)]) | Left (Con f g) <-
    sq, Left f `notElem` sq && Left g `notElem` sq]

simpleOr :: Rule (NestSeq MNNForm)
simpleOr (NS sq) = [("v", [NS (Left f : Left g : sq)]) | Left (Dis f g) <- sq, Left f
    `notElem` sq || Left g `notElem` sq]

toZipRule :: Rule (NestSeq f) -> Rule (ZipNestSeq f)
toZipRule ru (ZNS cs pa) = [(rule, [ZNS pm pa | NS pm <- premises]) | (rule, premises)
     <- ru (NS cs)]

childForm :: NestSeq MNNForm -> [MNNForm]
childForm (NS ns) = [a | Right (NS y) <- ns, Left a <- y]

diaK :: Rule (ZipNestSeq MNNForm)
```

```
diaK (ZNS cs (StepNS ls rs pa)) = [("<>", [ZNS (Left a : cs) (StepNS ls rs pa)]) |
    Left (Dia a) <- ls ++ rs, Left a `notElem` cs]
diaK (ZNS _ TopNS) = []

boxK :: Rule (ZipNestSeq MNNForm)
boxK (ZNS cs pa) = [("{}", [ZNS [Left a] (StepNS cs [] pa)]) | Left (Box a) <- cs, a `
    notElem` childForm (NS cs)]

safeK :: Rule (ZipNestSeq MNNForm)
safeK zns = concatMap ($ zns) [diaK, toZipRule simpleAnd, toZipRule simpleOr]

k :: Logic (ZipNestSeq MNNForm)
k = Log safeK boxK
```

To begin proof search, we convert the input formula into an open proof with only that formula as the root:

```
start :: f -> Proof f
start f = Node (ZNS [Left f] TopNS) "" []
```

We then continue proof search with `extend`, which takes a `Logic` and a `Proof` and extends it based on Algorithm 2. We focus on the case where the second argument has the form `Node (ZNS cs pa) "" []`, such as when we initialise a proof with `start`. It proceeds as follows:

1. check if the sequent is an axiom with `isAxiom`. If it is, close off the branch with `Proved` and terminate. Otherwise, move on to the next:

   ```
   isAtom :: MNNForm -> Bool
   isAtom (P _) = True
   isAtom _     = False

   isAxiom :: ZipNestSeq MNNForm -> Bool
   isAxiom (ZNS cs _) = any (\f -> isAtom f && neg f `elem` lefts cs) (lefts cs)
   ```

2. check if any `simpleRules` are applicable. If there is one, apply it and call `extend` on its premises. Otherwise, move on to the next:

3. check if any `boxRules` are applicable. If there is one, apply it and call `extend` on its premises. Otherwise, move on to the next:

4. check if the focus is at the root. If it is, terminate the proof search and leave the branch open. Otherwise, apply the `bk` rule to move the focus back to the parent, and call `extend`.

The function `extend` is defined as follows.

```
extend  :: Logic (ZipNestSeq MNNForm) -> Proof MNNForm -> Proof MNNForm
extend l (Node (ZNS cs pa) "" []) =
  case (isAxiom (ZNS cs pa), simpleRules l (ZNS cs pa), boxRules l (ZNS cs pa)) of
    (True, _, _)    -> Node (ZNS cs pa) "ax" [Proved]
    (_, rule:_, _) -> Node (ZNS cs pa) (fst rule) [extend l $ Node premise "" [] |
        premise <- snd rule]
    (_, [], rule:_) -> Node (ZNS cs pa) (fst rule) [extend l $ Node premise "" [] |
        premise <- snd rule]
    (_, [], [])     -> case pa of
      StepNS {} -> Node (ZNS cs pa) "bk" [extend l $ Node (move_parent $ ZNS cs pa) ""
          []]
      TopNS     -> Node (ZNS cs pa) "" []
```

The following cases are defined to complete pattern matching, and never occur if we apply `extend` to a proof initialised with `start`:

```
extend _ Proved = error "branch already closed"
extend _ (Node (ZNS _ _) (_:_) []) = error "no rule with no premise"
extend _ (Node (ZNS _ _) "" (_:_)) = error "can't have premise without rule name"
extend _ (Node (ZNS _ _) (_:_) (_:_)) = error "already extended"
```

We combine `start` with `extend` into `provek`, and check whether a formula is provable with `isProvablek` using `isClosedPf`:

```
provek :: MNNForm -> Proof MNNForm
provek = extend k . start

isProvablek :: MNNForm -> Bool
isProvablek = isClosedPf . provek
```

We define the function `ppProof` to pretty print a `Proof`. Note the output type `IO()`, as we directly print the output.

```
ppProof :: (Show f, PrettyPrint f) => Proof f -> IO ()
ppProof = ppp "" where
  ppp pref Proved = putStrLn $ pref ++ "Proved."
  ppp pref (Node zns rn ps) = do
    putStrLn (pref ++ pp zns)
    putStrLn (pref ++ rn)
    mapM_ (ppp (pref ++ "  ")) ps
```

For example, the following is the pretty printed output proof of the K axiom

```
ghci> ppProof (provek kaxiom)
ghci> ppProof ( provek kaxiom )
*(<>(P0 & ~P1) v (<>~P0 v {}P1))*
v
  *<>(P0 & ~P1), (<>~P0 v {}P1), (<>(P0 & ~P1) v (<>~P0 v {}P1))*
  v
    *<>~P0, {}P1, <>(P0 & ~P1), (<>~P0 v {}P1), (<>(P0 & ~P1) v (<>~P0 v {}P1))*
    {}
      <>~P0, {}P1, <>(P0 & ~P1), (<>~P0 v {}P1), (<>(P0 & ~P1) v (<>~P0 v {}P1)), *[P1
          ]*
      <>
        <>~P0, {}P1, <>(P0 & ~P1), (<>~P0 v {}P1), (<>(P0 & ~P1) v (<>~P0 v {}P1)),
            *[~P0, P1]*
        <>
          <>~P0, {}P1, <>(P0 & ~P1), (<>~P0 v {}P1), (<>(P0 & ~P1) v (<>~P0 v {}P1)),
              *[(P0 & ~P1), ~P0, P1]*
          &
            <>~P0, {}P1, <>(P0 & ~P1), (<>~P0 v {}P1), (<>(P0 & ~P1) v (<>~P0 v {}P1))
                , *[P0, (P0 & ~P1), ~P0, P1]*
            ax
              Proved.
            <>~P0, {}P1, <>(P0 & ~P1), (<>~P0 v {}P1), (<>(P0 & ~P1) v (<>~P0 v {}P1))
                , *[~P1, (P0 & ~P1), ~P0, P1]*
            ax
              Proved.
```

The following is the output of a failed proof search of the 4 axiom:

```
ghci> ppProof (provek fouraxiom)
*(<>~P0 v {}{}P0)*
v
```

```
*[<>~P0, {}{}P0, (<>~P0 v {}{}P0)]*
{}
  <>~P0, {}{}P0, (<>~P0 v {}{}P0), *[{}P0]*
  <>
    <>~P0, {}{}P0, (<>~P0 v {}{}P0), *[~P0, {}P0]*
    {}
      <>~P0, {}{}P0, (<>~P0 v {}{}P0), [~P0, {}P0, *[P0]*, ]
      bk
        <>~P0, {}{}P0, (<>~P0 v {}{}P0), *[~P0, {}P0, [P0]]*
```

Since Algorithm 2 works without having to store the history of the proof search, we also implement a more lightweight prover `extendNoHist` which does not build a `Proof`, but otherwise works analogously to `extend`:

```
extendNoHist :: Logic (ZipNestSeq MNNForm) -> ZipNestSeq MNNForm -> Bool
extendNoHist l (ZNS cs pa) =
  case (isAxiom (ZNS cs pa), simpleRules l (ZNS cs pa), boxRules l (ZNS cs pa)) of
    (True, _, _)    -> True
    (_, rule:_, _) -> all (extendNoHist l) (snd rule)
    (_, [], rule:_) -> all (extendNoHist l) (snd rule)
    (_, [], [])     -> case pa of
      StepNS {} -> extendNoHist l $ move_parent (ZNS cs pa)
      TopNS     -> False

isProvableNoHistk :: MNNForm -> Bool
isProvableNoHistk f = extendNoHist k (ZNS [Left f] TopNS)
```

In Section 5.5, we will test that `isProvablek`, which first generates a derivation using `extend` and check that the resulting proof is closed using `isClosedPf`, is equivalent to `isProvableNoHistk`.

5.4 Examples

In this section, we encode various formulas, which will be used to test our prover in Section 5.5. Some of these are schemas, taking in other formulas as arguments.

The following are valid formulas:

```
p, q, r :: MNNForm
p = P 0
q = P 1
r = P 2

lem :: MNNForm -> MNNForm
lem f = Dis f (neg f)

kaxiom :: MNNForm
kaxiom = Box (p `implies` q) `implies`  (Box p `implies` Box q)

kschema :: MNNForm -> MNNForm -> MNNForm
kschema f g = Box (f `implies` g) `implies`  (Box f `implies` Box g)

valid1 :: MNNForm -> MNNForm -> MNNForm
valid1 f g = Con (Box (f `implies` g)) (Box f) `implies` Box g

valid2 :: MNNForm -> MNNForm -> MNNForm
valid2 f g = Box (Con f g) `iff` Con (Box f) (Box g)

valid3 :: MNNForm -> MNNForm -> MNNForm
```

```
valid3 f g = Dia (Dis f g) `iff` Dis (Dia f) (Dia g)
```

The following are invalid formulas:

```
contradiction :: MNNForm -> MNNForm
contradiction f = Con f (neg f)

fouraxiom :: MNNForm
fouraxiom = Box p `implies` Box (Box p)

taxiom :: MNNForm
taxiom = Box p `implies` p

invalid1 :: MNNForm
invalid1 = Con (Dia p) (Dia q) `implies` Dia (Con p q)

invalid2 :: MNNForm
invalid2 = Box (Dis p q) `implies` Dis (Box p) (Box q)

glaxiom :: MNNForm
glaxiom = Box (Box p `implies` p) `implies` Box p
```

5.5 Simple Tests

We use the `Hspec` library to define tests to check the correctness our prover in Section 5.3. Some of the tests also make use of the `QuickCheck` library, generating random formulas as inputs using the `Arbitrary` instance of `MNNForm` defined in Section 5.1 The first test checks that `isProvableNoHistk` is equivalent to `isProvablek` on randomly generated input formulas:

```
main :: IO ()
main = hspec $ do
  describe "Testing prover" $ do
    prop "Provable with history iff provable without history" $
      \f -> isProvablek f == isProvableNoHistk f
```

The following tests check that valid formulas defined in Section 5.4 are provable.

```
  describe "Valid formulas are provable" $ do
    prop "Law of excluded middle" $
      isProvableNoHistk . lem
    prop "k axiom {}(A -> B) -> ({}A -> {}B)" $ do
      \f g -> isProvableNoHistk (kschema f g)
    prop "({}(A -> B) & {}A) -> {}B" $ do
      \f g -> isProvableNoHistk (valid1 f g)
    prop "{}(A & B) <-> {}A & {}B" $ do
      \f g -> isProvableNoHistk (valid2 f g)
    prop "<>(A v B) <-> <>A v <>B" $ do
      \f g -> isProvableNoHistk (valid3 f g)
```

The following tests check the invalid formulas are not provable.

```
  describe "Invalid formulas are not provable" $ do
    prop "A & ~A" $
      \f -> not $ isProvableNoHistk $ contradiction f
    it "4 axiom {}p ->{}{}p " $
      not $ isProvableNoHistk fouraxiom
    it "T axiom {}p -> p" $
      not $ isProvableNoHistk taxiom
```

```
    it "<>p & <>q -> <>(p & q)" $
      not $ isProvableNoHistk invalid1
    it "{}(p v q) -> {}p v {} q" $
      not $ isProvableNoHistk invalid2
    it "GL axiom {}({}p -> p) -> {}p" $
      not $ isProvableNoHistk glaxiom
```

The result of running `stack test` is as follows:

```
Testing prover
  Provable with history iff provable without history [v]
    +++ OK, passed 100 tests.
Valid formulas are provable
  Law of excluded middle [v]
    +++ OK, passed 100 tests.
  k axiom {}(A -> B) -> ({}A -> {}B) [v]
    +++ OK, passed 100 tests.
  ({}(A -> B) & {}A) -> {}B [v]
    +++ OK, passed 100 tests.
  {}(A & B) <-> {}A & {}B [v]
    +++ OK, passed 100 tests.
  <>(A v B) <-> <>A v <>B [v]
    +++ OK, passed 100 tests.
Invalid formulas are not provable
  A & ~A [v]
    +++ OK, passed 100 tests.
  4 axiom {}p ->{}{}p  [v]
  T axiom {}p -> p [v]
  <>p & <>q -> <>(p & q) [v]
  {}(p v q) -> {}p v {} q [v]
  GL axiom {}({}p -> p) -> {}p [v]

Finished in 0.0797 seconds
12 examples, 0 failures
```

Chapter 6

Conclusion and Future Work

After presenting the preliminaries, we examined a pre-existing proof search algorithm for modal logic K based on the nested sequent calculus NS_K. While simple in theory, we uncovered the implicit steps required when implementing the algorithm. Motivated by a heuristic to streamline the proof search procedure, we introduced nested sequents with focus, which augments the tree-like structure of nested sequents with a current position of interest. We presented the proof system NS^f_K, where the order of application of rules is enforced by the presence of the focus, accompanied by a proof search algorithm incorporating the heuristic, and showed that this system is equivalent to NS_K for derivability of formulas.

To demonstrate the benefit of the focused approach, we represent nested sequents with focus in Haskell using zippers, enabling efficient navigation through the structure, and succinctly implemented a prover which faithfully replicates the proof search algorithm of NS^f_K. To check the correctness of our prover, we tested it on simple formulas, where the variables are uniformly substituted by randomly generated formulas.

We hope that this thesis demonstrates the potential of zippers in both the theory and implementation of theorem proving. We conclude with a few directions for future work.

- The performance of our prover can be tested by benchmarking its run time with a set of formulas with a size parameter. Its performance can also be compared against other provers of modal logic K, such as those mentioned in Section 1.3.

- We can extend the system NS^f_K to other modal logics which can exploit the structure of nested sequents with focus to streamline the proof search procedure. The implementation has been designed to accommodate further extensions. Primary candidates include T and $\mathsf{S4}$, and the implementation has been designed to accommodate further extensions.

- The user experience of the prover can be enhanced. Currently, it is only operable with a Haskell installation. A web interface, along with a simplified syntax for inputting formulas can make the prover more accessible.

- The use of zippers for other logics and proof systems involving deep inference can be investigated.

Bibliography

[Brü09] Kai Brünnler. "Deep Sequent Systems for Modal Logic". In: *Archive for Mathematical Logic* 48.6 (July 2009), pages 551–577. ISSN: 1432-0665. https://doi.org/10.1007/s00153-009-0137-3 (cited on pages 2–4, 6, 8, 12).

[BRV10] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Fourth printing with corrections. Cambridge Tracts in Theoretical Computer Science 53. Cambridge: Cambridge University Press, 2010. ISBN: 978-1-107-05088-4. https://doi.org/10.1017/CBO9781107050884 (cited on page 6).

[Bul92] Robert A. Bull. "Cut Elimination for Propositional Dynamic Logic without *". In: *Mathematical Logic Quarterly* 38.1 (Jan. 1992), pages 85–100. ISSN: 0942-5616, 1521-3870. https://doi.org/10.1002/malq.19920380107 (cited on page 2).

[GS20] Marianna Girlando and Lutz Strassburger. "MOIN: A Nested Sequent Theorem Prover for Intuitionistic Modal Logics (System Description)". In: *IJCAR 2020 - 10th International Joint Conference*. July 2020, page 398. https://doi.org/10.1007/978-3-030-51054-1_25 (cited on page 3).

[Hue97] Gérard Huet. "The Zipper". In: *Journal of Functional Programming* 7.5 (Sept. 1997), pages 549–554. ISSN: 1469-7653, 0956-7968. https://doi.org/10.1017/S0956796897002864 (cited on pages 3, 9).

[Kas94] Ryo Kashima. "Cut-Free Sequent Calculi for Some Tense Logics". In: *Studia Logica* 53.1 (Mar. 1994), pages 119–135. ISSN: 1572-8730. https://doi.org/10.1007/BF01053026 (cited on page 2).

[Nv01] Sara Negri and Jan von Plato. *Structural Proof Theory*. Cambridge: Cambridge University Press, 2001. ISBN: 978-0-521-79307-0. https://doi.org/10.1017/CBO9780511527340 (cited on page 2).

[Pog09] Francesca Poggiolesi. "The Method of Tree-Hypersequents for Modal Propositional Logic". In: *Towards Mathematical Philosophy: Papers from the Studia Logica Conference Trends in Logic IV*. Edited by David Makinson, Jacek Malinowski, and Heinrich Wansing. Dordrecht: Springer, 2009, pages 31–51. ISBN: 978-1-4020-9084-4 (cited on page 2).

[van09] Twan van Laarhoven. *MOLTAP : A Modal Logic Tableau Prover*. Feb. 2009 (cited on page 3).

[Yan24] Xiaoshuang Yang. "Sequent Calculus with Zippers". Master's thesis. Universiteit van Amsterdam, 2024 (cited on pages 3, 22).