Sequent Calculus with Zippers

MSc Thesis *(Afstudeerscriptie)*

written by

Xiaoshuang Yang

under the supervision of **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.*

Institute for Logic, Language and Computation

Abstract

Sequent calculus, a method of formal logical argumentation developed by Gerhard Gentzen, is often used for backward-searching proofs. The success of sequent calculus is significantly influenced by the specific order and choice of inference rules and principal formulas used. The selection of appropriate rules and principal formulas in conventional sequent calculus provers often requires repeated trial and error, which is, however, a time-consuming and memory-intensive process.

To address this issue, in this thesis we develop a sequent calculus prover that employs zippers, a data structure introduced by Gérard Huet in 1997. Zippers can facilitate efficient navigation and modification of hierarchical data structures, such as trees and lists, enabling rapid, targeted updates without the necessity for complete reconstruction. The use of zippers enables the dynamic management of the proof tree, thereby facilitating the application of different rules to specific sequents while maintaining the integrity of the remainder of the proof tree.

We investigate whether the use of zippers can enhance the efficiency of the proof search process and reduce the requisite computational resources. We developed two generic modular provers. One uses the conventional tree representation, and another uses a zipper. For both provers, we then implement the three systems G3C, G3I, and G3K. Subsequently, we make a comparison of their efficiency in terms of run time and memory usage.

Our findings indicate that in some cases where the formula is unprovable, the zipper-based prover consumes a markedly reduced amount of time and memory compared to the tree-based prover. This improvement can be attributed to the fact that the zipper-based prover does not (or almost does not) have to perform garbage collection, which optimises the proof search process by efficiently handling the unfocused components of the proof tree.

Contents

1 Introduction

Sequent calculus, developed by Gerhard Gentzen, is a method of formal logical argumentation. It is particularly useful for backwards searching proofs. However, the effectiveness of this method, especially in intuitionistic logic and modal logic, depends heavily on the precise order and choice of inference rules as well as active formulas. Conventional sequence calculus provers often require repeated trial and error to select the correct rules and active formulas and to find the optimal order in which to apply the rules, and this demands considerable time and memory.

To address this problem, my thesis aims to develop a sequent calculus prover that uses zippers, a data structure introduced by Gérard Huet [Hue97] in 1997. Zippers enable efficient navigation and modification of hierarchical data structures such as trees and lists. They allow for fast, targeted updates without complete reconstruction. By using zippers, the proof tree can be dynamically managed, meaning that we can simply apply different rules to the focused sequents and keep the rest of the proof tree. Our research question is this:

> Will using zippers make the proof search process more efficient and reduce the required computational resources significantly?

We implement two generic modular provers in Haskell for each of the systems **G3C**, **G3I** and **G3K**. One prover for each logic uses the conventional tree representation. Another will use a Zipper. We then compare their efficiency in terms of memory use and run time.

Chapter 2 In this chapter we introduce the concept of zippers and explore their application to lists, trees and proof trees. We provide a detailed explanation of how zippers work and why they are useful for efficiently managing hierarchical data structures.

Chapter 3 This chapter covers the logics we will implement: classical propositional logic, intuitionistic propositional logic, and minimal modal logic K. We also introduce the notion of provers and the sequent calculus as a proof system, providing the theoretical background for our implementations.

Chapter 4 Here we describe in detail our Haskell implementation of the generic modular provers. One prover uses the conventional tree representation, while the other uses a zipper. We discuss the design and architecture of these implementations and how they address the challenges of sequent calculus proof search.

Chapter 5 In this chapter, we discuss the three sequent calculus systems used and their corresponding Haskell implementations: G3C for classical propositional logic, G3I for intuitionistic propositional logic, and G3K for minimal modal logic K. We provide detailed descriptions of each system and how they are encoded in Haskell.

Chapter 6 This chapter presents the formulas and schemata used for tests and benchmarks.

Chapter 7 In this chapter we present the results of tests and benchmarks. We compare the two provers in terms of run time and memory usage when proving the same formulas. We discuss the observed performance differences and analyse the reasons for the efficiency of the zipper-based prover, especially in cases of unprovable formulas constructed using `foldr`.

Chapter 8 We conclude with possibilities for future work. Given more time, we could implement additional modal logics using our modular provers. In addition, zippers could be very useful for falsifying proof systems such as tableau systems. Based on our results, we discuss possible directions for further research and improvements.

Here, we provide all the Haskell code used in this thesis in the following GitHub repository: https://github.com/XiaoshuangYang999/Sequent-Calculus-With-Zippers.git .

2 Zipper

The zipper is a family of data structures that allows traversal and modification of another aggregate data structure while maintaining the context of the position within the structure. Essentially, it provides a means to navigate and edit elements of a data structure while efficiently keeping track of the location. The zipper achieves this by "splitting" the structure into two parts: the focused part (where the current operation is happening) and the rest of the structure, often referred to as the context or the path.

This concept was introduced by Gérard Huet in his seminal 1997 paper "Functional Pearl: The Zipper." Huet's work presented the zipper as a technique for managing pointers in a functional setting.

Huet describes the zipper as follows:

> "The tree is turned inside-out like a returned glove, pointers from the root to the current position being reversed in a path structure. The current location holds both the downward current subtree and the upward path. All navigation and modification primitives operate on the location structure. Going up and down in the structure is analogous to closing and opening a zipper in a piece of clothing, whence the name." [Hue97]

The zipper technique can be used for many recursively defined data structures, like lists and trees. In this thesis, we will elaborate on how to read and understand zippers for lists, trees, and proof trees in the context of Haskell. We will also cover navigate and mutate functions such as move left, move right, move up, move down, delete, insert, change, and the translation between zipped and unzipped structures.

2.1 Zipper for Lists

In Haskell, lists are a homogeneous data structure, meaning they are ordered containers that store multiple objects of the same type.

```
-- The empty list
emptyList :: [()]
emptyList = []

-- A string : a list of characters
aString :: String
aString = "HelloWorld"
-- We can also define lists of Integers, Booleans ...
```

Since a list is ordered, what if we want to have a focus on the list and perform local changes? Below we define a class for the navigation and modification functions we want [Gat23a]:

```
class ListLike l where
-- | Create a zipper with a single item.
  singleton :: a -> l a
-- | Get the item at the current point.
  get :: l a -> a
-- | Move left, assuming we are not at the head already.
  moveLeft :: l a -> l a
-- | Move right, assuming we are not at the end already.
  moveRight :: l a -> l a
-- | Insert an item afer the current point.
  insertAfter :: a -> l a -> l a
-- | Delete item at current point, assuming it is not the last.
  delete :: l a -> l a
```

One way is to create a new data structure consisting of a list and an integer that represents the location:

```
data LocList a = Loc [a] Int
```

However, this approach only tells us the location of our focus. It still does not make it easy to change the value at the focus, and the memory usage is higher.

```
instance ListLike LocList where
  singleton x = Loc [x] 0
  get (Loc xs x) = xs !! x
  moveLeft (Loc _ 0 ) = error "Cannot move left!"
  moveLeft (Loc xs x ) = Loc xs (x - 1)
  moveRight (Loc xs x) = if x + 1 == length xs
                         then error "Cannot move right"
                         else Loc xs (x + 1)
  insertAfter w (Loc xs x) = Loc (take (x + 1) xs ++ [w] ++ drop (x + 1) xs) x
  delete (Loc xs x) = if x + 1 == length xs
                         then error "Cannot delete last element"
                         else Loc (take x xs ++ drop (x + 1) xs) x
```

Instead, we can use a zipper for lists:

```
-- zipper for lists
data ZipList a = Zip [a] a [a]

ziplExample :: ZipList Char
ziplExample = Zip "olleH" 'W' "orld"

loclExample :: LocList Char
loclExample = Loc "HelloWorld" 5
```

Next, we create a `ListLike` instance for `ZipList`:

```
instance ListLike ZipList where
  singleton x = Zip [] x []
  get (Zip _ p _) = p
  moveLeft (Zip [] _ _ ) = error "Cannot move left!"
  moveLeft (Zip (x:xs) p ys) = Zip xs x (p:ys)
  moveRight (Zip _  _ []     ) = error "Cannot move right!"
  moveRight (Zip xs p (y:ys)) = Zip (p:xs) y ys
  insertAfter w (Zip xs x ys) = Zip xs x (w:ys)
  delete (Zip _  _ []     ) = error "Cannot delete last element!"
  delete (Zip xs _ (y:ys)) = Zip xs y ys
```

Finally, here is the translation between `ZipList` and `LocList`

```
fromZipL :: ZipList a -> LocList a
fromZipL (Zip ys x xs) = Loc (reverse ys ++ [x] ++ xs) (length ys)

toZipL :: LocList a -> ZipList a
toZipL (Loc xs x) = Zip (reverse (take x xs)) (xs!!x) (drop (x + 1) xs)
```

2.2 Zipper for Trees

Trees are hierarchical and non-linear data structures, making them more complex than lists. They consist of nodes connected by edges, with each node containing a value and pointers to its children. The Zipper structure can be effectively applied to trees to allow for efficient traversal and modification while keeping track of the context.

The following is a standard definition of trees in Haskell

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

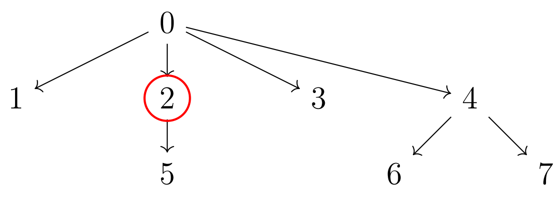
The zipper for trees works similarly to the zipper for lists. It focuses on a particular subtree while retaining the path to reach its root.

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

data ZipTree a = ZT (Tree a) (Path a) -- location
```

In the `Path` type, the first argument `a` of `Step a (Path a) [Tree a] [Tree a]` refers to the parent node of the current node. The next argument, of type `Path a`, indicates the path of the parent node. The first list refers to the list (ordered) of the right siblings of the current node, and the second list refers to the left siblings.

To make this more concrete, let's look at some examples:

Example 2.1. Here is an easy example for a `ZipTree`:

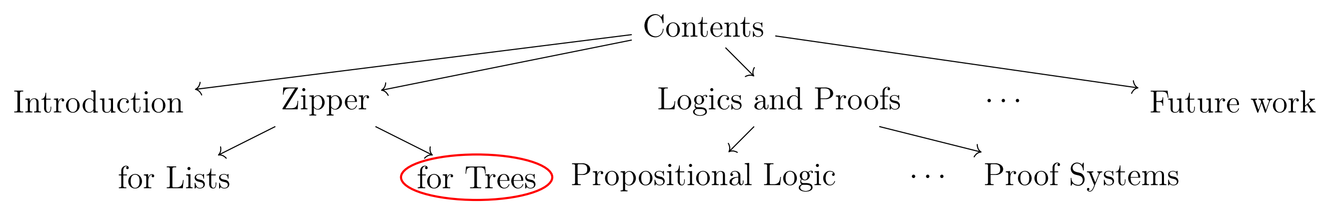


The following is the code for this tree:

```
aZipTree :: ZipTree Int
aZipTree = ZT (Node 2 [Node 5 []]) (Step 0 Top [Node 1[]] [Node 3 [], Node 4 [Node 6[],
    Node 7[]]])
```

Here, we focus on the subtree `Node 2 [Node 5 []]`. 0 is its parent node, and it's the root of the whole tree. The subtree has one right sibling, `Node 1 []`, and two left siblings, `Node 3 []` and `Node 4 [Node 6[], Node 7[]]`.

Example 2.2. We use our current location in the thesis as a more complex example.



The following is the Haskell code for it:

```haskell
ourLocation :: ZipTree String
ourLocation = ZT
    (Node "for Trees" [])
    (Step "Zipper"
          (Step "Contents" Top
                [Node "Introduction" []]
                [Node "Logics and Proofs"
                        [Node "Propositional Logic" []
                        ,Node "Modal Logic" []
                        ,Node "Automated Theorem Prover" []
                        ,Node "Proof Systems" []]
                ,Node "Haskell Implementation: The General Prover"
                        [Node "Sequent" []
                        ,Node "Proof" []
                        ,Node "Tree-Based Prover and Zipper-Based Prove" []
                        ,Node "The Tree-Based Prover" []
                        ,Node "The Zipper-Based Prover" []]
                ,Node "Haskell Implementation: The Logics"
                        [Node "G3C for Classical Propositional Logic" []
                        ,Node "G3I for Intuitionistic Propositional Logic" []
                        ,Node "G3K for Minimal Modal Logic" []]
                ,Node "Tests and Bencmarks"
                        [Node "Formulas" []
                        ,Node "Additional Correctness Tests" []]
                ,Node "Results"
                        [Node "Test Results" []
                        ,Node "Run Time" []
                        ,Node "Memory Usage" []
                        ,Node "Discussion" []]
                ,Node "Future Work" []
                ]
          )
          [Node "for Lists" []]
          [])
```

Similarly with `ListLike`, we define a class `TreeLike` for all the navigation and modifications we want. (We will also use this class for proof trees). We only want the class to contain the minimal functions we want

```haskell
class TreeLike z where
  zsingleton :: a -> z a
  move_left :: z a -> z a
  move_right :: z a -> z a
  move_up :: z a-> z a
  move_down :: z a -> z a
  zdelete :: z a -> z a
```

Now we can add a `TreeLike` instance for `ZipTree`:

```haskell
instance TreeLike ZipTree where
  zsingleton x = ZT (Node x []) Top
  move_left (ZT c (Step s p (x:xs) ys)) = ZT x (Step s p xs (c:ys))
  move_left _ = error "cannot go left"
  move_right (ZT c (Step s p xs (y:ys))) = ZT y (Step s p (c:xs) ys)
  move_right _ = error "cannot go right"
  move_up (ZT c (Step s p xs ys)) = ZT (Node s ((c:xs) ++ ys)) p
  move_up _ = error "cannot go up"
  move_down (ZT (Node s (x:xs)) p) = ZT x (Step s p [] xs)
  move_down _ = error "cannot go down"
  zdelete (ZT _ (Step x p xs ys)) = ZT (Node x (xs++ys)) p
  zdelete _ = error "cannot delete top"
```

Here is the translation between zipped and unzipped Trees

```
fromZipT :: ZipTree a -> Tree a
fromZipT (ZT t Top) = t
fromZipT (ZT t (Step x p xs ys)) = fromZipT $ ZT (Node x (xs ++ (t:ys))) p

toZipT :: Tree a -> ZipTree a
toZipT t = ZT t Top
```

Later, in Chapter 4, we will introduce zipper for proof trees.

3 Logics and Proofs

In this chapter, we will cover the logics we will implement: classical propositional logic, intuitionistic propositional logic, and minimal modal logic K. We will also introduce the notion of provers and proof systems, especially sequent calculus used in our implementation.

3.1 Propositional Logic

Definition 3.1 (The language $\mathcal{L}_{\text{PL}}$). We fix a list of symbols and the well-formed formulas of $\mathcal{L}_{\text{PL}}$ are given by:

$$\varphi ::= \bot \mid \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \to \varphi \mid \varphi \leftrightarrow \varphi$$

Haskell Implementation of $\mathcal{L}_{\text{PL}}$

The following Haskell code defines $\mathcal{L}_{\text{PL}}$. We define formulas of $\mathcal{L}_{\text{PL}}$ to be of type `FormP`. Notice that in our implementation, we choose $\bot, \wedge, \vee, \to$ to be our primitive symbols and follow proof theory's convention to define $\top, \neg, \leftrightarrow$[TS00]:

$$\begin{aligned} \top &:= \bot \to \bot \\ \neg\varphi &:= \varphi \to \bot \\ \varphi \leftrightarrow \psi &:= (\varphi \to \psi) \wedge (\psi \to \varphi) \end{aligned}$$

```
type Atom = Char
data FormP = BotP | AtP Atom | ConP FormP FormP | DisP FormP FormP | ImpP FormP FormP
  deriving (Eq,Ord)

topP :: FormP
topP = Imp BotP BotP

negP :: FormP -> FormP
negP f = ImpP f BotP

iffP :: FormP -> FormP -> FormP
iffP f g = ConP (ImpP f g) (ImpP g f)
```

Now we can define a `Show` instance for `FormP` so we can print it.

```
instance (Show FormP) where
  show BotP       = "⊥"
  show (AtP a)    = [a]
  show (ConP f g) = "(" ++ show f ++ " ∧ " ++ show g ++ ")"
  show (DisP f g) = "(" ++ show f ++ " v " ++ show g ++ ")"
  show (ImpP f g) = "(" ++ show f ++ " → " ++ show g ++ ")"
```

We also create an `Arbitrary` instance for `FormP` so we can generate random formulas to perform tests.

```
instance Arbitrary FormP where
  arbitrary = sized genForm where
    factor = 2
    genForm 0 = oneof [ pure BotP, AtP <$> choose ('p','t') ]
    genForm 1 = AtP <$> choose ('p','t')
    genForm n = oneof
      [ pure BotP
      , AtP <$> choose ('p','t')
      , ImpP <$> genForm (n `div` factor) <*> genForm (n `div` factor)
      , ConP <$> genForm (n `div` factor) <*> genForm (n `div` factor)
      ]
```

Classical Propositional Logic and Intuitionistic Propositional Logic

The difference between classical propositional logic (CPL) and intuitionistic propositional logic (IPL) lies in the number of valid formulas, with IPL having fewer valid formulas. From the perspective of proof theory, CPL can be seen as an extension of IPL. Specifically, CPL can be viewed as IPL plus one of the following schemata: the law of excluded middle, double negation, or Peirce's law.

3.2 Modal Logic

Definition 3.2 (The language $\mathcal{L}_{\mathrm{ML}}$)**.** We fix a list of symbols and the well-formed formulas of $\mathcal{L}_{\mathrm{ML}}$ are given by:

$$\varphi ::= \bot \mid \top \mid p \mid \neg\varphi \mid \varphi\wedge\varphi \mid \varphi\vee\varphi \mid \varphi\to\varphi \mid \varphi\leftrightarrow\varphi \mid \Box\varphi \mid \Diamond\varphi$$

Haskell Implementation of $\mathcal{L}_{\mathrm{ML}}$

The following Haskell code defines $\mathcal{L}_{\mathrm{ML}}$. We define formulas of $\mathcal{L}_{\mathrm{ML}}$ to be of type `FormM`.

Notice that in our implementation, we choose $\bot, \wedge, \vee, \to, \Box$ to be our primitive symbols and view $\Diamond$ as the dual of $\Box$.

$$\Diamond\varphi := \neg\Box\neg\varphi$$

```
data FormM = BotM | AtM Atom | ConM FormM FormM | DisM FormM FormM | ImpM FormM FormM | Box
    FormM
  deriving (Eq,Ord)
```

```
negM :: FormM -> FormM
negM f = ImpM f BotM

topM :: FormM
topM = negM BotM

iffM :: FormM -> FormM -> FormM
iffM f g = ConM (ImpM f g) (ImpM g f)

diaM :: FormM -> FormM
diaM f = negM $ Box $ negM f
```

We also define `Show` instance and `Arbitrary` instance for `FormM`.

```
instance (Show FormM) where
  show BotM        = "⊥"
  show (AtM a)     = [a]
  show (ConM f g)  = "(" ++ show f ++ " ∧ " ++ show g ++ ")"
  show (DisM f g)  = "(" ++ show f ++ " v " ++ show g ++ ")"
  show (ImpM f g)  = "(" ++ show f ++ " → " ++ show g ++ ")"
  show (Box f)    = "(" ++ " □ " ++ show f ++ ")"

instance Arbitrary FormM where
  arbitrary = sized genForm where
    factor = 2
    genForm 0 = oneof [ pure BotM, AtM <$> choose ('p','t') ]
    genForm 1 = AtM <$> choose ('p','t')
    genForm n = oneof
      [ pure BotM
      , AtM <$> choose ('p','t')
      , ImpM <$> genForm (n `div` factor) <*> genForm (n `div` factor)
      , ConM <$> genForm (n `div` factor) <*> genForm (n `div` factor)
      , Box <$> genForm (n `div` factor)
      ]
```

We know that all formulas of $\mathcal{L}_{\text{PL}}$are also formulas of $\mathcal{L}_{\text{ML}}$, since modal logic is an extension of propositional logic. But now ML formulas and PL formulas have different types. So we create a translation function here.

```
pTom :: FormP -> FormM
pTom BotP = BotM
pTom (AtP x) = AtM x
pTom (ConP x y) = ConM (pTom x) (pTom y)
pTom (DisP x y) = DisM (pTom x) (pTom y)
pTom (ImpP x y) = ImpM (pTom x) (pTom y)
```

The Minimal Modal Logic K

Minimal modal logic, often called modal logic K, is one of the foundational systems in the study of modal logic. Introduced by Saul Kripke in the 1960s, the logic extends classical propositional logic by incorporating modal operators $\Box$ and $\Diamond$. The semantics of modal logic are defined in terms of Kripke models, which consist of possible worlds and reachability relations between them. A formula $\Box\varphi$ is considered true if it holds in all its reachable worlds. For more information, please see [BDRV01].

The most typical tautology in K is called the K axiom, whence the name of minimal modal logic, where φ,ψ can be any modal formulas

$$\Box(\varphi \to \psi) \to (\Box\varphi \to \Box\psi)$$

3.3 Automated Theorem Prover

An automated theorem prover, or prover for short, is a computer program designed to verify the validity of logical statements. Provers work by applying a set of inference rules to derive conclusions from given premises. They explore possible logical derivations until a proof is found or the statement is deemed unprovable.

Provers can be implemented using a variety of programming languages and paradigms. For example, the LoTREC system, written in Java, is a generic tableau theorem prover for modal logic, which can build models or counter-models for formulas [GHLS19].On the other hand, writing provers in functional languages such as Haskell could have great potential because of their strong support for recursion and higher-order functions, which are well suited to logical operations. For example, SMCDEL [Gat24] is a symbolic model checker for Dynamic Epistemic Logic written in Haskell.

In practice, provers are used for formal verification of software and hardware, to ensure that systems behave as expected by their specifications. They are also used for reasoning tasks in artificial intelligence, for automated reasoning in mathematical theorem proving, and in academic research to explore new logical systems and their properties.

3.4 Proof Systems

Proof systems are formal frameworks used to derive logical conclusions from a set of axioms and inference rules. Proofs are typically presented as trees, which are constructed according to the axioms and rules of inference of a given logical system.

Sequent Calculus

Sequent calculus, developed by Gerhard Gentzen in 1933, is a proof system particularly useful for backward searching proofs. For a general intro, refer to [TS00]. The Gentzen-style calculi have many variants for propositional logic, such as one-sided, G1-calculi, G2-calculi, G3-calculi, and more. For simplicity in implementation, we use the G3-calculi. The most significant feature of G3-calculi is that it incorporates structural rules (like contraction and weakening) into the logical rules.

In G3-calculi, we derive sequents, which are expressions like $\Gamma \Rightarrow \Delta$, where Γ and Δ are finite multisets of formulas. This expression should be understood as $\bigwedge \Gamma \to \bigvee \Delta$.

The proof in G3-calculi is defined as follows [TS00]:

> Proofs are labelled finite trees with a single root, with axioms at the top nodes, and each node label connected with the labels of the (immediate) successor nodes (if any) according to one of the rules. The rules are divided into left- (L-) and right- (R-) rules. For a logical operator $\bigotimes$ say, L$\bigotimes$, R$\bigotimes$ indicate the rules where a formula with $\bigotimes$ as main operator is introduced on the left and on the right respectively.

Definition 3.3 (G3C)**.** G3C, is the sequent calculus for classical propositional logic:

$$(\mathrm{L}\bot)\,\frac{}{\Gamma, \bot \Rightarrow \Delta} \qquad (\mathrm{Ax})\,\frac{}{\Gamma, p \Rightarrow \Delta, p}$$

$$(\mathrm{L}\wedge)\,\frac{\Gamma, \alpha, \beta \Rightarrow \Delta}{\Gamma, \alpha \wedge \beta \Rightarrow \Delta} \qquad (\mathrm{R}\wedge)\,\frac{\Gamma \Rightarrow \Delta, \alpha \qquad \Gamma \Rightarrow \Delta, \beta}{\Gamma \Rightarrow \Delta, \alpha \wedge \beta}$$

$$(\mathrm{L}\vee)\,\frac{\Gamma, \alpha \Rightarrow \Delta \qquad \Gamma, \beta \Rightarrow \Delta}{\Gamma, \alpha \vee \beta \Rightarrow \Delta} \qquad (\mathrm{R}\vee)\,\frac{\Gamma \Rightarrow \Delta, \alpha, \beta}{\Gamma \Rightarrow \Delta, \alpha \vee \beta}$$

$$(\mathrm{L}\!\to)\,\frac{\Gamma, \beta \Rightarrow \Delta \qquad \Gamma \Rightarrow \Delta, \alpha}{\Gamma, \alpha \to \beta \Rightarrow \Delta} \qquad (\mathrm{R}\!\to)\,\frac{\Gamma, \alpha \Rightarrow \Delta, \beta}{\Gamma \Rightarrow \Delta, \alpha \to \beta}$$

Below is a proof of Pierce's Law in G3C.

Example 3.1. $\vdash_{G3C} ((p \to q) \to p) \to p$

$$(\mathrm{R}\!\to)\,\dfrac{(\mathrm{L}\!\to)\,\dfrac{(\mathrm{Ax})\,\dfrac{}{p \Rightarrow p} \qquad (\mathrm{Ax})\,\dfrac{}{p \Rightarrow p, q}}{(p \to q) \to p \Rightarrow p}}{\Rightarrow ((p \to q) \to p) \to p}$$

We will introduce two more G3-calculi in Chapter 5. They are G3I for intuitionistic propositional logic and G3K for the minimal modal logic K.

Other proof systems, equivalent to sequent calculus, include Hilbert systems, natural deduction, and Tableau.

4 Haskell Implementation: The General Prover

In this chapter, we explain the Haskell implementation of our two sequent calculus provers, with standard trees and with zippers. Both provers are generic in the sense that they are not defined for a specific logic. We then specify concise implementations for classical propositional logic, intuitionistic propositional logic and modal logic in Chapter 3.

4.1 Sequent

We assume `f` to be a formula type. We have defined two concrete formula types in Chapter 3: `FormP` for propositional logic and `FormM` for modal logic.

In sequent calculus, a sequent has the form of $\Gamma \Rightarrow \Delta$, where Γ and Δ are multisets of formulas. We now need to choose the Haskell type of these multisets. We have three data structures from different libraries to choose from: `List` (from `Data.List`), `MultiSet` (from `Data.MultiSet`), and `Set` (from `Data.Set`).

List The simplest option. However, in our prover, we frequently need to check sublist relations and membership. Lists have `isSubsequenceOf` and `elem` operations, but membership checking in lists has a time complexity of $O(n)$, and the time complexity for `isSubsequenceOf` is $O(n \cdot m)$ for `isSubsequenceOf xs ys` where `length xs = m` and `length ys = n`. .

MultiSet More efficient for membership checking due to tree-like storage structures, but checking sublist relations is more complex and time-consuming due to multiple occurrences of the same element. Additionally, `Data.MultiSet` provides fewer functions.

Set Provides $O(\log n)$ complexity for membership checking with `member`, similar to MultiSet, but without the complexities involved in sublist checking. It has a built-in function `isSubsetOf` with time complexity $O(m \log(\frac{n}{m} + 1))$ for `isSubsetOf xs ys` where `length xs = m` and `length ys = n`..

In the logics we choose here, the multiplicity of formulas in sequents does not affect the derivations. So we can choose the `Set` type for representing multisets in sequent calculus. Using the Gentzen style of sequent calculus, we have formulas on the left and right sides of the implication. A straightforward representation could be:

```
data Sequent' f = S (Set f) (Set f)
```

However, in sequent calculus, left implication moves formulas from left to right and right implication moves formulas from right to left, which complicate the implementation.

A more convenient method is to tag each formula with a position label using Haskell type `Either`, which simplifies manipulations and provides helper functions.

Thus, we define a sequent as:

```
type Sequent f = Set (Either f f)
```

We define functions to print a sequent in the usual form, separating the set into the left and right sides:

```
-- helper functions
leftsSet :: Ord a => Set (Either a a) -> Set a
leftsSet xs = Set.map fromEither $ Set.filter isLeft xs

rightsSet :: Ord a => Set (Either a a) -> Set a
rightsSet xs = Set.map fromEither $ Set.filter isRight xs
```

```
-- pretty printing a list of formulas
ppList :: Show f => [f] -> String
ppList [] = ""
ppList [f] = show f
ppList (f:fs@(_:_)) = show f ++ " , " ++ ppList fs

-- pretty printing a set of formulas
ppForm :: Show f => Set f -> String
ppForm ms = ppList (Set.toList ms)

-- pretty printing a sequent of formulas
ppSeq :: (Show f, Ord f) => Sequent f -> String
ppSeq xs = ppForm (leftsSet xs) ++ " => " ++ ppForm (rightsSet xs)
```

The output will look like:

```
-- ghci> ppSeq . Set.fromList $ [Left (AtP 'p'), Left (AtP 'q' ), Right (AtP 'r' )]
-- "p , q => r"
```

4.2 Proof

Now that we have sequents, we can define proofs and print them.

First, we have a string as `RuleName`, for example, like $R \rightarrow$, $L\vee$.

Since sequent calculus is particularly useful for backtracking proofs, our prover will prove formulas backwards. This means the root of the proof tree will be the sequent we aim to prove. Although the sequent calculus rule is technically applied to the children sequents, resulting in the parent sequent, for simplicity, we will describe the process as applying the "rule" from the parent to the children. Thus, the second "rule" should be considered the inverse of the actual sequent calculus rule.

In this implementation, when we refer to the "parent" sequent, we mean the resulting sequent after applying a sequent calculus rule. Conversely, the "child" sequent refers to the preceding sequent in the application of the rule.

A proof of formula type `f` can be of two forms:

`Proved` indicating this branch is proved/closed, similar to "Q.E.D." used in mathematical proofs.

`Node (Sequent f) RuleName [Proof f]` a tree with a sequent as its root, a list of proofs as its children, and a `RuleName` indicating the rule/axiom applied to go from the parent to the children.

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

There cannot be children without rules, and there cannot be rules without children.

In all versions of sequent calculus we will use, in a proof, every branch can have at most two children. We also create a function to check this condition to ensure that the output proofs are indeed well-defined proofs for our systems.

```
hasLeqTwoChildren :: Eq f => Proof f -> Bool
hasLeqTwoChildren Proved = True
hasLeqTwoChildren (Node _ _ ts) = length ts <= 2 && all hasLeqTwoChildren ts
```

Based on the structure, we can define a function `isClosedPf` that determines whether a proof is closed:

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

The visualization of proofs is done via the Haskell library Graphviz [MS23]. And we use the DispAble class defined in [Gat23b].

```
class DispAble t where
  toGraph :: t -> DotM String ()
  disp :: t -> IO ()
  disp x = runGraphvizCanvas Dot (digraph' $ toGraph x) Xlib
  dot :: t -> IO ()
  dot x = graphvizWithHandle Dot (digraph' $ toGraph x) Canon $ \h -> do
    hSetEncoding h utf8
    SB.hGetContents h >>= SB.putStr
  svg :: t -> String
  svg x = unsafePerformIO $ withSystemTempDirectory "tapdleau" $ \tmpdir -> do
    _ <- runGraphvizCommand Dot (digraph' $ toGraph x) Svg (tmpdir ++ "/temp.svg")
    readFile (tmpdir ++ "/temp.svg")
  pdf :: t -> IO FilePath
  pdf x = runGraphvizCommand Dot (digraph' $ toGraph x) Pdf "temp.pdf"
```

```
instance (Show f,Ord f) => (DispAble (Proof f)) where
  toGraph = toGraph' "" where
    toGraph' pref Proved =
      node pref [shape PlainText, toLabel "□"]
    toGraph' pref (Node fs rule' ts) = do
      node pref [shape PlainText, toLabel $ ppSeq fs]
      mapM_ (\(t,y') -> do
        toGraph' (pref ++ show y' ++ ":") t
        edge pref (pref ++ show y' ++ ":") [toLabel rule']
        ) (zip ts [(0::Integer)..])
```

4.3 Tree-Based Prover and Zipper-Based Prover

Now that we have sequents and proofs defined, we can introduce our two sequent calculus provers based on different types: one on ProofWithH, and one on ZipProof.

The type ProofWithH is a pair of a list of sequents and a proof. The first element is the "history," which includes all the sequents seen before in the process of developing this proof. It's designed for history-tracking and mainly designed to prove formulas in intuitionistic logic. If we get to a point where one of the new sequents has been seen before, then we know we have entered a loop, so we need to terminate the prover.

```
type ProofWithH f = ([Sequent f], Proof f)
```

In contrast, ZipProof is a proof along with a path. Similar to Path in Chapter 2, the ZipPath indicates the location of the focused subtree: it includes an additional argument RuleName, which indicates the rule applied from the parent of the focused subtree's root to the root.

```
data ZipPath f = Top | Step (Sequent f) RuleName (ZipPath f) [Proof f] [Proof f]

data ZipProof f = ZP (Proof f) (ZipPath f)
```

Notice that the first RuleName in Proof is the rule applied from the root to its children, whereas the RuleName in the path is the rule applied from the parent of the root to the root.

Now we can define the type of rules for these two methods. We will have RuleT f and RuleZ f.

For either rule type, it's a function that takes two arguments: one is of type ProofWithH f / ZipProof f, which is the current proof to be extended; the second argument is a tagged principal formula, which we assume is an element of the sequent we are extending.

After receiving these inputs, the function of type `RuleT/RuleZ` will return `[[(RuleName,[Sequent f])]]`.

```
type RuleT f = ProofWithH f -> Either f f -> [[ (RuleName,[Sequent f])]]

type RuleZ f = ZipProof f -> Either f f -> [[ (RuleName,[Sequent f])]]
```

To understand the necessity of the second argument, it's important to recognize that in all the logics discussed in this thesis, given a fixed principal formula (or formulas), there is only one rule that corresponds to this formula (or formulas).

But this is not the case in other proof systems. For example, in G3S[TS00], the sequent calculus system for modal logic S4 has rule K□ and R□:

$$(\mathrm{K}\Box)\,\frac{\Sigma \Rightarrow \beta}{\Gamma, \Box\Sigma \Rightarrow \Box\beta, \Delta} \qquad (\mathrm{R}\Box)\,\frac{\Box\Sigma \Rightarrow \beta}{\Gamma, \Box\Sigma \Rightarrow \Box\beta, \Delta}$$

The resulting sequents in K□ and R□ have exactly the same structures and the same principal formulas. But their previous sequents are not the same. As we are proving formulas backwards, we can only know the sequents below and identify the principal formulas among them.

We explain the result type of `RuleT` and `RuleZ` in three parts:

Outer most brackets The outermost brackets function like the "Maybe" type, indicating that if there is an applicable rule for this formula (in the sense that a rule in the sequent calculus system is found and the condition-checking is satisfied), then the output will be a singleton of type `[(RuleName, [Sequent f])]`. If no applicable rule is found, the output will be an empty list.

We use these brackets instead of `Maybe` because it can be dangerous to extract values from `Maybe`. The function `fromJust` may return an error if the argument is `Nothing`.

The middle brackets The middle brackets indicate all possible ways of applying the same rule. This is only used in modal logic, where we have a rule $K\Box$:

$$(\mathrm{K}\Box)\,\frac{\Sigma \Rightarrow \beta}{\Gamma, \Box\Sigma \Rightarrow \Box\beta, \Delta}$$

Even if we specify the right box formula as the right principal formula, there are still multiple ways of choosing the principal formulas on the left side, resulting in different possible sequents from the same sequent. This is what we mean by "different possible ways of applying the same rule." For CPL and IPL, this layer of the list will always be a singleton.

The pair `(RuleName, [Sequent f])` The first element is the name of the rule we find applicable to the given input. The second element is a list of resulting children after the rule application. The brackets represent branching, which happens in $R\wedge$, $L\vee$, $L \rightarrow$.

Now, we can define what a logic is. A logic for formula type `f` needs to have:

- `neg`: negation
- `bot`: Bottom
- `isAtom`: to determine whether a formula is an atom
- `isAxiom`: to determine whether a sequent is an axiom
- Safe and unsafe rule
  - `safeRuleT`: a rule function for `ProofWithH`. Here, for safe rule, we mean all the invertible rules in sequent calculus. These rules will only have one principal formula in the resulting sequent/ the parent sequent
  - `unsafeRuleT`: a list of rule functions for `ProofWithH`. In our actual usage, it will either be empty or a singleton. By unsafe rules, we mean those rules that will lose information after being applied, for example, R$\rightarrow$ in G3I and K$\Box$ in G3K
  - Similarly for `safeRuleZ` and `unsafeRuleZ`

```
data Logic f = Log
  { neg :: f -> f
  , bot :: f
  , isAtom :: f -> Bool
  , isAxiom :: Sequent f -> Bool
  , safeRuleT :: RuleT f
  , unsafeRuleT :: [RuleT f]
  , safeRuleZ :: RuleZ f
  , unsafeRuleZ :: [RuleZ f]
  }
```

4.4 The Tree-Based Prover

4.4.1 Preliminary

Suppose we want to prove a formula φ. In sequent calculus, we always start by building a sequent $\Rightarrow \varphi$ and try to prove it backwards.

We use `startForT` to build a "start point": an incomplete proof with an empty history:

```
startForT :: f -> ProofWithH f
startForT f =  ([],Node (Set.singleton (Right f)) "" [])
```

Now we define some helper functions for our main proving function `extendT`.

`isClosedPfT` will determine whether a given `ProofWithH` is closed. A proof with history is closed if the proof itself is closed.

`isApplicableToT` will determine whether a rule is applicable for a proof with history and a principal formula, by applying the arguments to the rule function. It is applicable as long as the output is not an empty list.

```
isClosedPfT :: Eq f => ProofWithH f -> Bool
isClosedPfT (_,fs) = isClosedPf fs

isApplicableToT :: ProofWithH f -> Either f f -> RuleT f -> Bool
isApplicableToT fs f r = not . List.null $ r fs f
```

4.4.2 ExtendT

Now we are ready to define our extendT function. The goal of extendT is to extend the “start point” to a list of complete proofs. For a closed branch, the leaf will be Proved, and for an open branch, the leaf will be a sequent without children.

`extendT` takes two arguments: the first is a logic, and the second is a proof with history. The output will be a list of all possible proofs with their history.

We mainly focus on the case where the second argument is of the form `pt@(h, Node fs "" [])`, which is the only case used in the proving process. Other cases are defined to avoid warnings.

Given a logic `l` and a `ProofWithH` `pt@(h, Node fs "" [])`, the program checks four things sequentially:

1. Is there a left bottom in `fs`?

   If so, we can simply output `[(h, Node fs "L⊥" [Proved])]`.

   If not, we will move to the next question.

2. Is `fs` an axiom?

   If so, we can simply output `[(h, Node fs "Ax" [Proved])]`.

   If not, we will move to the next question.

3. Is any formula in `fs` applicable to the safe rule function of logic `l`?

   If so, `lookupMin` picks a formula that satisfies this condition. Although minimality is not required, it helps in picking any applicable formula. If such a formula is found, we extend the proof by applying the rule to each child sequent in the `result`. This creates a list of lists of completed proofs with history, which we combine using `pickOneOfEach` to get all possible proofs. Each proof is then added as a child of the original sequent with the rule name `therule`. If no such formula exists, we proceed to the next question.

   ```
   pickOneOfEach :: [[a]] -> [[a]]
   pickOneOfEach [] = [[]]
   pickOneOfEach (l:ls) = [ x:xs | x <- l, xs <- pickOneOfEach ls ]
   ```

4. Does logic `l` have an unsafe rule?

   If not, then we reach a dead end. We stop the proving process.

   If yes, we check if any formula in `fs` is applicable to this rule. If not, the sequent is unprovable. If such formulas exist, we apply the rule and extend the new sequent. For each resulting proof, we append it as a child of the original proof tree. We combine all possible ways to get a list of all completed proofs with history. If any proof is closed, we return the first such proof. Otherwise, the sequent is unprovable.

```
extendT  :: (Eq f, Show f, Ord f) => Logic f -> ProofWithH f -> [ProofWithH f ]
extendT l pt@(h, Node fs "" [])=
  case ( Left (bot l) ‘Set.member‘ fs,
        isAxiom l fs,
        Set.lookupMin $ Set.filter (\g -> isApplicableToT pt g (safeRuleT l)) fs,
        unsafeRuleT l
        )  of
```

```
  (True,_, _,_ )    -> [(h, Node fs "L⊥" [Proved])]
  -- fs is an axiom
  (_,True, _,_ )    -> [(h, Node fs "Ax" [Proved])]
  -- The safe rule can be applied
  (_   ,_,Just f,_ )   -> [
                  (h, Node fs therule (List.map snd ts) ) |
                  (therule,result) <- head $ safeRuleT l pt f,
                  ts <- pickOneOfEach [ extendT l (fs : h, Node nfs "" [])| nfs <- result
                      ]
                  ]
  -- The logic has no unsafe rule -> CPL
  (_   ,_,Nothing,[])  -> [(h, Node fs "" [])]
  -- The logic has an unsafe rule -> IPL, K
  (_   ,_,Nothing,r:_) -> case checkEmpty $ Set.filter (\g -> isApplicableToT pt g r) fs of
        -- No applicable formulas in fs
        Nothing -> [(h, Node fs "" [])]
        -- fs contains applicable formulas
        Just gs ->  if any isClosedPfT nps
                      then List.take 1 (List.filter isClosedPfT nps)
                      else [(h, Node fs "" [])]
          where
            nps = concat $ List.concatMap tryExtendT gs
            -- tryExtendT :: Either f f -> [[([Sequent f], Proof f)]]
            tryExtendT g = [  List.map (\pwh -> (h, Node fs therule [snd pwh]))
                                $ extendT l (fs : h, Node (head result) "" [])
                            | (therule,result) <- head $ (head.unsafeRuleT $ l) pt g ]
-- just for pattern matching
extendT l (h,Node fs r@(_:_) xs@(_:_))= [ (h,Node fs r $ List.map snd nfs)
                                         | nfs <- List.map (\f -> extendT l (fs:h,f)) xs]
extendT _ (h,Proved) = [(h,Proved)]
extendT _ (_,Node _ (_:_) [])= error"cannot have rules and no children"
extendT _ (_,Node _ [] (_:_)) = error"cannot have children and no rules"
```

4.4.3 Provability and Visualization

Now we have the `extendT` function. We can use it to actually prove things. In `proveT` we first use `startForT` to build the original `ProofWithH` for the formula and then feed it to the `extendT` function. Eventually, it transforms all the resulting `ProofWithH` into proofs.

Then `isProvableT` will output whether the formula f is provable or not, based on whether there are closed proofs among those proofs we found through `proveT l f`.

```
proveT :: (Eq f, Show f,Ord f) => Logic f -> f -> [Proof f]
proveT l f = List.map snd $ extendT l (startForT f)

isProvableT :: (Eq f, Show f, Ord f) => Logic f -> f -> Bool
isProvableT l f = any isClosedPf (proveT l f)
```

The function `provePrintH` will print the proof we found. For the provable formula, the proof will be closed. For the unprovable formula, the proof won't be closed.

The function `provePdfH` is using the method `pdf` in the `Displayable` class. `provePdfH l f` will create a "temp.pdf" of the proof.

```
proveprintT :: (Eq f, Show f,Ord f) => Logic f -> f -> Proof f
proveprintT l f = if isProvableT l f
                  then head $ List.filter isClosedPf (proveT l f)
                  else head (proveT l f)

provePdfT :: (Ord f,Show f, Eq f) => Logic f -> f -> IO FilePath
provePdfT l f= pdf $ proveprintT l f
```

4.5 The Zipper-Based Prover

4.5.1 Helper Functions

Remember that in Chapter 2, we define a class `TreeLike`:

```
class TreeLike z where
  zsingleton :: a -> z a
  move_left :: z a -> z a
  move_right :: z a -> z a
  move_up :: z a-> z a
  move_down :: z a -> z a
  zdelete :: z a -> z a
```

Now we add a `TreeLike` instance for `ZipProof`:

```
instance TreeLike ZipProof where
  zsingleton x = ZP (Node (Set.singleton (Right x)) "" []) Top
  move_left (ZP c (Step s r p (x:xs) ys)) = ZP x (Step s r p xs (c:ys))
  move_left _ = error "cannot go left"
  move_right (ZP c (Step s r p xs (y:ys))) = ZP y (Step s r p (c:xs) ys)
  move_right _ = error "cannot go right"
  move_up (ZP c (Step s r p xs ys)) = ZP (Node s r ((c:xs) ++ ys)) p
  move_up _ = error "cannot go up"
  move_down (ZP (Node s r (x:xs)) p) = ZP x (Step s r p [] xs)
  move_down _ = error "cannot go down"
  zdelete (ZP _ (Step s _ Top _ _)) = ZP (Node s "" []) Top
  zdelete (ZP _ (Step s _ p _ _)) = ZP (Node s "" []) p
  zdelete _ = error "cannot delete top"
```

We also define a function that transforms a `ZipProof` into `Proof` so that we can reuse the `DispAble` instance for `Proof`:

```
fromZip :: ZipProof f -> Proof f
fromZip (ZP x Top) = x
fromZip zp = fromZip (move_up zp)
```

4.5.2 The Proving Process

We introduce the order of extending proofs for a zip type (left-biased). Since we are implementing sequent calculus here, every node has at most two branches.

- When we reach a node and find a rule to apply, we move to and try to extend its left child (by method `move_down`).
- Whenever we find a dead end, we initially try to move to its right sibling. If there isn't any, we move to its parent. If its parent has a right sibling, we move there. If not, we move up again. We call this switch process function `switch`.

```
-- A function that tells whether a node has right siblings
hasRsibi :: ZipPath f -> Bool
hasRsibi (Step _ _ _ _ (_:_))= True
hasRsibi _ = False

-- Switch path, left-biased
switch :: ZipProof f -> ZipProof f
switch (ZP pf Top) = ZP pf Top
switch (ZP pf p) = if hasRsibi p
```

```
                       then move_right (ZP pf p)
                       else switch.move_up $ ZP pf p
```

The proving logic behind `extendZ` is essentially the same as `extendT`, except for different methods for manipulating data structures. We gightlight the differences in the comments in the code below.

```
startForZ :: f -> ZipProof f
startForZ f = ZP (Node (Set.singleton (Right f)) "" []) Top

isClosedZP :: Eq f => ZipProof f -> Bool
isClosedZP (ZP fs Top) = isClosedPf fs
isClosedZP (ZP fs p) = isClosedPf fs &&  (isClosedZP . switch $ ZP fs p)

isApplicableToZ :: ZipProof f -> Either f f -> RuleZ f -> Bool
isApplicableToZ fs f r = not . List.null $ r fs f

extendZ  :: (Ord f,Eq f) => Logic f -> ZipProof f -> [ZipProof f]
extendZ l zp@(ZP (Node fs "" []) p) =
  case ( Left (bot l) `Set.member` fs,
        isAxiom l fs,
        Set.lookupMin $ Set.filter (\g -> isApplicableToZ zp g (safeRuleZ l)) fs,
        unsafeRuleZ l)  of
  -- Switch the path if the current sequent is closed
  (True,_,_   ,_ )     -> extendZ l (switch (ZP (Node fs "L⊥" [Proved]) p))
  (_,True,_   ,_ )     -> extendZ l (switch (ZP (Node fs "Ax" [Proved]) p))
  -- Find a safe rule to use
  (_ ,_   ,Just f,_)      -> extendZ l (move_down $ ZP (Node fs therule ts) p) where
            (therule,result) = head . head $ safeRuleZ l zp f
            ts = [ Node nfs "" []| nfs <- result]
  -- Check if there is unsafe rule
        -- Whenever a dead end is found, stop the proving process
  (_ ,_   ,Nothing ,[])     -> [ZP (Node fs "" []) p]
        -- Has an unsafe rule
  (_,_    ,Nothing ,r:_)    -> case  checkEmpty $ Set.filter (\g -> isApplicableToZ zp g r)
      fs of
                -- Not applicable
                Nothing -> [ZP (Node fs "" []) p]
                -- Find a list of formulas to apply
                Just gs -> if any isClosedZP nps
                             then List.take 1 (List.filter isClosedZP nps)
                             else [ZP (Node fs "" []) p]
                       where
                         nps = concat $ List.concatMap tryExtendZ gs
                         tryExtendZ g = [ extendZ l (ZP (Node (head result) "" []) (Step fs
                             therule p [] []) )
                           | (therule,result) <- head $ (head.unsafeRuleZ $ l) zp g ]
extendZ _ (ZP Proved p) = [ZP Proved p]
extendZ _ (ZP (Node _ (_:_) []) _ )= error"cannot have rules and no children"
extendZ _ (ZP (Node fs r@(_:_) xs@(_:_)) p )= [ZP (Node fs r xs) p]
extendZ _ (ZP (Node _ [] (_:_)) _) = error"cannot have children and no rules"
```

Similarly, we can define provability check functions and visualization functions.

```
proveZ :: (Eq f, Ord f) => Logic f -> f -> [Proof f]
proveZ l f = List.map fromZip $ extendZ l (startForZ f)

isProvableZ :: (Eq f, Ord f) => Logic f -> f -> Bool
isProvableZ l f = any isClosedZP $ extendZ l (startForZ f)

proveprintZ :: (Eq f, Ord f) => Logic f -> f -> Proof f
proveprintZ l f = if isProvableZ l f
                then head $  List.filter isClosedPf (proveZ l f)
                else head (proveZ l f)

provePdfZ :: (Show f, Eq f,Ord f) => Logic f -> f -> IO FilePath
provePdfZ l f = pdf $ proveprintZ l f
```

4.5.3 Comparison

We highlight three kinds of differences between these two proves:

- **Data Structure:**
  - `extendT` operates on `ProofWithH`, which uses lists to keep track of history and current proof state.
  - `extendZ` operates on `ZipProof`, which uses a zipper data structure for navigation and updates within the proof tree.
- **Traversal and Backtracking:**
  - `extendT` uses straightforward list operations for history and backtracking.
  - `extendZ` leverages the zipper's ability to move left, right, up, and down.
- **Implementation Complexity:**
  - `extendT` was simpler to implement due to the use of basic list operations.
  - `extendZ` is more complex but provides a cleaner and more modular approach to navigating and modifying the proof tree.

5 Haskell Implementation: The Logics

In this section, we will explain the Haskell implementation for the three logics introduced in Chapter 3. First, we need to choose a sequent calculus system for each logic and then implement the rules in Haskell. We choose G3C for classical propositional logic, G3I for intuitionistic logic, and G3K for minimal modal logic.

5.1 G3C for Classical Propositional Logic

Here we implement classical propositional logic using the G3C system introduced in definition 3.3.

Remember, a logic is constructed as follows:

```haskell
data Logic f = Log
  { neg :: f -> f
  , bot :: f
  , isAtom :: f -> Bool
  , isAxiom :: Sequent f -> Bool
  , safeRuleH :: RuleH f
  , unsafeRuleH :: [RuleH f]
  , safeRuleZ :: RuleZ f
  , unsafeRuleZ :: [RuleZ f]
  }
```

To define CPL, we already defined negation and bottom for FormP in Chapter 3. We still need to define isAtom and isAxiom before implementing the rules.

We define isatomP and isAxiomP and will use them both in classical and intuitionistic propositional logic.

```
isatomP :: FormP -> Bool
isatomP (AtP _) = True
isatomP _ = False

isAxiomP :: Sequent FormP -> Bool
isAxiomP fs = any (\f -> isatomP f && Right f ‘Set.member‘ fs) (leftsSet fs)
```

Now we can implement the connective rules. The function safeCPL takes a labelled formula and returns a list of type [(RuleName, [Sequent FormP])]. This list is either empty, meaning the labelled formula is a labelled atomic formula, or a singleton of a pair, where the first element of the pair is the name of the rule and the second one is the resulting list of sequents after applying this rule.

```
safeCPL :: Either FormP FormP -> [(RuleName,[Sequent FormP])]
safeCPL (Left (ConP f g))   = [("L∧", [Set.insert (Left g) $ Set.singleton (Left f) ]     )]
safeCPL (Left (DisP f g))   = [("L∨", [Set.singleton (Left f)   , Set.singleton (Left g)] )]
safeCPL (Left (ImpP f g))   = [("L→", [Set.singleton (Left g)   , Set.singleton (Right f)])]
safeCPL (Right (ConP f g))  = [("R∧", [Set.singleton (Right f)  , Set.singleton (Right g)])]
safeCPL (Right (DisP f g))  = [("R∨", [Set.insert (Right g) $ Set.singleton (Right f) ]  )]
safeCPL (Right (ImpP f g))  = [("R→", [Set.insert (Right g) $ Set.singleton (Left f)])]
safeCPL _                   = []
```

We now define two helper functions that transform a function of type

Either f f -> [(RuleName, [Sequent f])] into RuleH f and RuleZ f. We will use these again when implementing G3K.

These functions take a function fun (such as safeCPL for FormP), an argument of the form (_, Node fs "" []) or ZP (Node fs "" []) _, and a formula g of type Either f f. They compute the new child sequents by combining the context formulas in fs with the new formulas generated by fun.

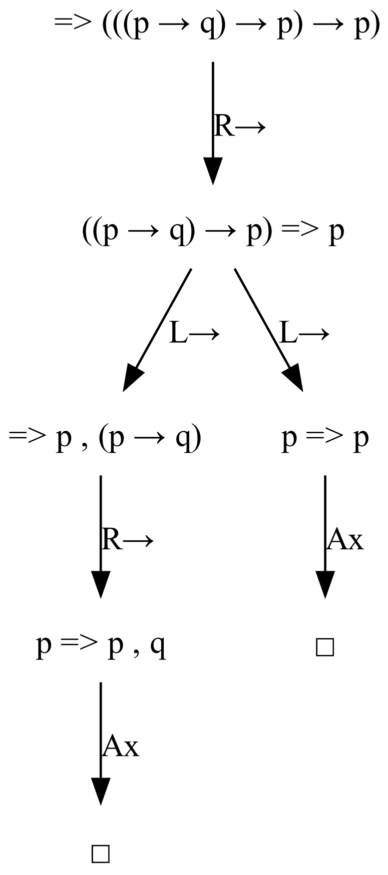
To notice: we always require g to be a member of fs when using replaceRuleT and replaceRuleZ.

```
replaceRuleZ :: (Eq f,Ord f) => (Either f f -> [(RuleName,[Sequent f])]) -> RuleZ f
replaceRuleZ fun (ZP (Node fs "" []) _) g =
    [[(fst . head $ fun g
    ,[ Set.delete g fs ‘Set.union‘ newfs | newfs <- snd . head $ fun g])]
    | not (List.null (fun g))]
replaceRuleZ _ _ _ = []

replaceRuleT :: (Eq f,Ord f) => (Either f f -> [(RuleName,[Sequent f])]) -> RuleH f
replaceRuleT fun (_,Node fs "" []) g =
    [[(fst . head $ fun g
    ,[ Set.delete g fs ‘Set.union‘ newfs | newfs <- snd . head $ fun g])]
    | not (List.null (fun g))]
replaceRuleT _ _ _ = []
```

Since all the rules in G3C are invertible, G3C has no unsafe rules. We now define classical as:

```
classical :: Logic FormP
classical = Log
  { neg = negP
  , bot = BotP
  , isAtom = isatomP
  , isAxiom = isAxiomP
  , safeRuleT = replaceRuleT safeCPL
  , unsafeRuleT = []
  , safeRuleZ = replaceRuleZ safeCPL
  , unsafeRuleZ = []
  }
```

Example 5.1. Here we present the identical proof of Pierce's law generated by our two provers, which is the same as in Example 3.1



5.2 G3I for Intuitionistic Propositional Logic

Many calculi implement intuitionistic propositional logic. However, most calculi require that the succedent be either empty or a singlet. This setting is different from the one we choose for classical propositional logic and would be harder to implement.

Here we choose a variant of the system called Maehara's Calculus, m-G3i. It was originally proposed in [Mae54], but here we follow [Dyc16]. It is known as Maehara's multi-succedent version m-G3i, where the only non-invertible rule is R→, and all indeterminacy during the proof search lies in this rule. This makes it easier to implement. Some of its original rules, such as L→, have built-in contraction (keeping a copy of the original formula in the new sequence), and others don't. Even if rules like L→ are invertible, they can still cause the program to lead to a loop without detecting repeated formulas. So we need to check the saturation condition. The easier way to do this is to add built-in contraction to every rule and perform the saturation check in every step.

Definition 5.1 (G3I)**.** We use G3I to denote our variant of m-G3i.

$$(\mathrm{L}\bot)\ \frac{}{\Gamma, \bot \Rightarrow \Delta} \qquad (\mathrm{Ax})\ \frac{}{\Gamma, p \Rightarrow \Delta, p}$$

$$(\mathrm{L}\wedge)\ \frac{\Gamma, \alpha, \beta, \alpha \wedge \beta \Rightarrow \Delta}{\Gamma, \alpha \wedge \beta \Rightarrow \Delta} \qquad (\mathrm{R}\wedge)\ \frac{\Gamma \Rightarrow \Delta, \alpha, \alpha \wedge \beta \qquad \Gamma \Rightarrow \Delta, \beta, \alpha \wedge \beta}{\Gamma \Rightarrow \Delta, \alpha \wedge \beta}$$

$$(\mathrm{L}\vee)\ \frac{\Gamma, \alpha \vee \beta, \alpha \Rightarrow \Delta \qquad \Gamma, \alpha \vee \beta, \beta \Rightarrow \Delta}{\Gamma, \alpha \vee \beta \Rightarrow \Delta} \qquad (\mathrm{R}\vee)\ \frac{\Gamma \Rightarrow \Delta, \alpha, \beta, \alpha \vee \beta}{\Gamma \Rightarrow \Delta, \alpha \vee \beta}$$

$$(\mathrm{L}\to)\ \frac{\Gamma, \beta, \alpha \to \beta \Rightarrow \Delta \qquad \Gamma, \alpha \to \beta \Rightarrow \Delta, \alpha}{\Gamma, \alpha \to \beta \Rightarrow \Delta} \qquad (\mathrm{R}\to)\ \frac{\Gamma, \alpha \Rightarrow \beta, \alpha \to \beta}{\Gamma \Rightarrow \Delta, \alpha \to \beta}$$

Similarly as `safeCPL`, we define `safeIPL` and `unsafeIPL` for R→.

```
safeIPL :: Either FormP FormP -> [(RuleName,[Sequent FormP])]
safeIPL (Left (ConP f g))   = [("L∧", [Set.insert (Left g) $ Set.singleton (Left f) ]    )]
safeIPL (Left (DisP f g))   = [("Lv", [Set.singleton (Left f)  , Set.singleton (Left g)] )]
     -- branch
safeIPL (Right (ConP f g))  = [("R∧", [Set.singleton (Right f) , Set.singleton (Right g)])]
     -- branch
safeIPL (Right (DisP f g))  = [("Rv", [Set.insert (Right g) $ Set.singleton (Right f) ]  )]
safeIPL (Left (ImpP f g))   = [("L→", [Set.singleton (Left g)  , Set.singleton (Right f)])]
     -- branch
safeIPL _                   = []

unsafeIPL :: Either FormP FormP -> [(RuleName,[Sequent FormP])]
unsafeIPL (Right (ImpP f g)) = [("R→", [Set.insert (Right g) $ Set.singleton (Left f)])]
unsafeIPL  _                 = []
```

To see if a rule is applicable, we also need to check two additional things:

- Saturation check: if all the resulting principal formulas in one of the resulting sequents are already in the current sequent, we call this situation saturated, and we do not apply this rule. Since we have built-in contraction here, if we do not detect saturation, we may end up in a loop of repeatedly adding the same formulas to the sequents, which is a complete waste of time.
- History check: we check whether applying a rule will get us any sequent seen in the history. This is because if we have seen this sequent before, we are not making any process of applying this rule. This check will efficiently speed up our algorithm.

Both checks are implemented in Haskell as follows:

```
saturated :: Sequent FormP -> Either FormP FormP -> Bool
saturated fs f@(Right (ImpP _ _)) =  any (`Set.isSubsetOf` fs) (snd . head . unsafeIPL $ f)
saturated fs f                    =  List.null (safeIPL f)
                                  || any (`Set.isSubsetOf` fs) (snd . head . safeIPL $ f)
```

```
-- return true iff has appeared before
historySearch :: ZipPath FormP -> Sequent FormP -> Bool
historySearch Top _ = False
historySearch (Step xs _ p _ _) ys = Set.isSubsetOf ys xs || historySearch p ys

-- return true iff doesn't result in loop
historyCheckZ :: ZipPath FormP -> Sequent FormP -> Either FormP FormP -> Bool
historyCheckZ p fs f@(Right (ImpP _ _)) = not (historySearch p (head xs)) where
                                   xs = applyIPL fs f (snd(head(unsafeIPL f)))
historyCheckZ _ _ _ = False

historyCheckT :: [Sequent FormP] -> Sequent FormP -> Either FormP FormP -> Bool
historyCheckT hs fs f@(Right (ImpP _ _)) = not $ any (Set.isSubsetOf (head xs)) hs where
                                     xs = applyIPL fs f (snd(head(unsafeIPL f)))
historyCheckT _ _ _ = False
```

Unlike G3C and G3K, which are both systems without built-in contraction, G3I has a built-in contraction. That is to say, we must retain a copy of the original principal formula after applying the rule. So we need a different version of `replaceRule` functions.

For the safe rules, We define `replaceRuleIPLsafe` and `replaceRuleIPLsafeT`; and for R→, we define `replaceRuleIPLunsafe` and `replaceRuleIPLunsafeT`.

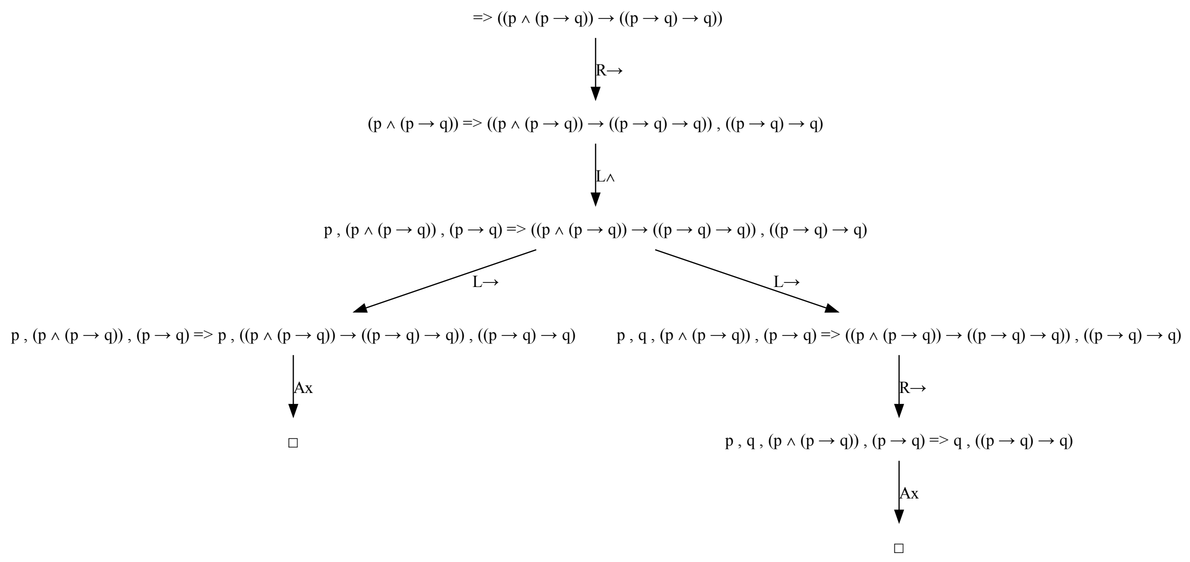
```
replaceRuleIPLsafeZ :: (Either FormP FormP -> [(RuleName,[Sequent FormP])]) -> RuleZ FormP
replaceRuleIPLsafeZ fun (ZP (Node fs "" []) _) g = [[(fst . head $ fun g
                                                    ,[ fs `Set.union` newfs | newfs <- snd .
                                                         head $ fun g])]
                                                   | not (saturated fs g)
                                                   && not (List.null (fun g))
                                                   ]
replaceRuleIPLsafeZ _ _ _= []

replaceRuleIPLsafeT :: (Either FormP FormP -> [(RuleName,[Sequent FormP])]) -> RuleT FormP
replaceRuleIPLsafeT fun (_,Node fs "" []) g = [[(fst . head $ fun g
                                              ,[ fs `Set.union` newfs | newfs <- snd .
                                                   head $ fun g])]
                                             | not (saturated fs g)
                                             && not (List.null (fun g))
                                             ]
replaceRuleIPLsafeT _ _ _= []

-- helper function for replaceRuleIPLunsafeZ and replaceRuleIPLunsafeT
applyIPL :: Sequent FormP -> Either FormP FormP -> [Sequent FormP] -> [Sequent FormP]
applyIPL fs f = List.map (Set.insert f (leftOfSet fs) `Set.union`)
-- leftOfSet = Set.filter isLeft

replaceRuleIPLunsafeZ :: (Either FormP FormP -> [(RuleName,[Sequent FormP])]) -> RuleZ
    FormP
replaceRuleIPLunsafeZ fun (ZP (Node fs "" []) p) g = [[(fst . head $ fun g
                                                      , applyIPL fs g (snd(head(unsafeIPL g)
                                                          ))) ]
                                                     | not (saturated fs g)
                                                     && historyCheckZ p fs g
                                                     && not (List.null (fun g))
                                                     ]
replaceRuleIPLunsafeZ _ _ _= []

replaceRuleIPLunsafeT :: (Either FormP FormP -> [(RuleName,[Sequent FormP])]) -> RuleT
    FormP
replaceRuleIPLunsafeT fun (h,Node fs "" []) g = [[(fst . head $ fun g
                                                 , applyIPL fs g (snd(head(unsafeIPL g))))
                                                     ]
                                                | not (saturated fs g)
```

```
                                                    && historyCheckT h fs g
                                                    && not (List.null (fun g))
                                                    ]
replaceRuleIPLunsafeT _ _ _= []
```

Finally, we define our logic `intui` as follows:

```
intui :: Logic FormP
intui = Log
  { neg = negP
  , bot = BotP
  , isAtom = isatomP
  , isAxiom = isAxiomP
  , safeRuleT = replaceRuleIPLsafeT safeIPL
  , unsafeRuleT = [replaceRuleIPLunsafeT unsafeIPL]
  , safeRuleZ = replaceRuleIPLsafeZ safeIPL
  , unsafeRuleZ = [replaceRuleIPLunsafeZ unsafeIPL]
  }
```

Example 5.2. Here we present the identical proof of $(p \wedge (p \to q)) \to ((p \to q) \to q)$ generated by our two provers.



=> ((p ∧ (p → q)) → ((p → q) → q))
R→
(p ∧ (p → q)) => ((p ∧ (p → q)) → ((p → q) → q)) , ((p → q) → q)
L∧
p , (p ∧ (p → q)) , (p → q) => ((p ∧ (p → q)) → ((p → q) → q)) , ((p → q) → q)
L→
L→
p , (p ∧ (p → q)) , (p → q) => p , ((p ∧ (p → q)) → ((p → q) → q)) , ((p → q) → q)
p , q , (p ∧ (p → q)) , (p → q) => ((p ∧ (p → q)) → ((p → q) → q)) , ((p → q) → q)
Ax
R→
□
p , q , (p ∧ (p → q)) , (p → q) => q , ((p → q) → q)
Ax
□

5.3 G3K for Minimal Modal Logic

Here we implement the minimal modal logic using the system called "G3K" in [TS00].

Definition 5.2 (G3K)**.** G3K is essentially the extension of G3C plus the modal rule K□.

$$(\text{L}\bot)\,\frac{}{\Gamma, \bot \Rightarrow \Delta} \qquad (\text{Ax})\,\frac{}{\Gamma, p \Rightarrow \Delta, p}$$

$$(\text{L}\wedge)\,\frac{\Gamma, \alpha, \beta \Rightarrow \Delta}{\Gamma, \alpha \wedge \beta \Rightarrow \Delta} \qquad (\text{R}\wedge)\,\frac{\Gamma \Rightarrow \Delta, \alpha \qquad \Gamma \Rightarrow \Delta, \beta}{\Gamma \Rightarrow \Delta, \alpha \wedge \beta}$$

$$(\mathrm{L}\vee)\ \frac{\Gamma, \alpha \Rightarrow \Delta \qquad \Gamma, \beta \Rightarrow \Delta}{\Gamma, \alpha \vee \beta \Rightarrow \Delta} \qquad (\mathrm{R}\vee)\ \frac{\Gamma \Rightarrow \Delta, \alpha, \beta}{\Gamma \Rightarrow \Delta, \alpha \vee \beta}$$

$$(\mathrm{L}\rightarrow)\ \frac{\Gamma, \beta \Rightarrow \Delta \qquad \Gamma \Rightarrow \Delta, \alpha}{\Gamma, \alpha \rightarrow \beta \Rightarrow \Delta} \qquad (\mathrm{R}\rightarrow)\ \frac{\Gamma, \alpha \Rightarrow \Delta, \beta}{\Gamma \Rightarrow \Delta, \alpha \rightarrow \beta}$$

$$(\mathrm{K}\Box)\ \frac{\Sigma \Rightarrow \beta}{\Gamma, \Box\Sigma \Rightarrow \Box\beta, \Delta}$$

First, we define `isAtomM` and `isAxiomM` for `FormM`. We need to do it again because `FormM` is not `FormP`.

```
isatomM :: FormM -> Bool
isatomM (AtM _) = True
isatomM _ = False

isAxiomM :: Sequent FormM -> Bool
isAxiomM fs = any (\f -> isatomM f && Right f ‘Set.member‘ fs) (leftsSet fs)
```

Then we implement the propositional rules:

```
safeML :: Either FormM FormM -> [(RuleName,[Sequent FormM])]
safeML (Left (ConM f g))  = [("L∧", [Set.insert (Left g) $ Set.singleton (Left f) ]     )]
safeML (Left (DisM f g))  = [("Lv", [Set.singleton (Left f)  , Set.singleton (Left g)] )]
safeML (Left (ImpM f g))  = [("L→", [Set.singleton (Left g)  , Set.singleton (Right f)])]
safeML (Right (ConM f g)) = [("R∧", [Set.singleton (Right f) , Set.singleton (Right g)])]
safeML (Right (DisM f g)) = [("Rv", [Set.insert (Right g) $ Set.singleton (Right f) ]  )]
safeML (Right (ImpM f g)) = [("R→", [Set.insert (Right g) $ Set.singleton (Left f)])]
safeML _                  = []
```

We first define some helper functions that manipulate `Box` inside `Either`.

```
isLeftBox :: Either FormM FormM -> Bool
isLeftBox (Left (Box _)) = True
isLeftBox _              = False

isRightBox :: Either FormM FormM -> Bool
isRightBox (Right (Box _)) = True
isRightBox _               = False

fromBox :: Either FormM FormM -> Either FormM FormM
fromBox (Left (Box f)) = Left f
fromBox g = g

removeBoxLeft :: Sequent FormM -> Sequent FormM
removeBoxLeft xs = Set.map fromBox $ Set.filter isLeftBox xs
```

K□ is a rule where, even if we fix a sequent and the principal formula on the right, the results may vary because we also need to select the principal formulas on the left. All other formulas will be deleted in the resulting sequent. Consequently, we need to generate the powerset of box formulas on the left.

```
func :: FormM -> Sequent FormM -> [(RuleName,[Sequent FormM])]
func f fs = [ ("K□", [Right f ‘Set.insert‘ fs]) ]

kboxT :: RuleT FormM
kboxT (_,Node fs "" []) (Right (Box f)) = Set.toList $ Set.map (func f) $ Set.powerSet .
    removeBoxLeft $ fs
kboxT _ _ =[]

kboxZ :: RuleZ FormM
kboxZ (ZP (Node fs "" []) _) (Right (Box f)) = Set.toList $ Set.map (func f) $ Set.powerSet
    .removeBoxLeft $ fs
kboxZ _ _ =[]
```

Now we can define our modal logic:

```
modal :: Logic FormM
modal = Log
  { neg = negM
  , bot = BotM
  , isAtom = isatomM
  , isAxiom = isAxiomM
  , safeRuleT = replaceRuleT safeML
  , unsafeRuleT = [kboxT]
  , safeRuleZ = replaceRuleZ safeML
  , unsafeRuleZ = [kboxZ]
  }
```

Example 5.3. Here we present the identical proof of K axiom generated by our provers.

$$\begin{tikzcd}
& {=> ((\Box (a \to b)) \to ((\Box a) \to (\Box b)))} \arrow[d, "R\to"] & \\
& {(\Box (a \to b)) => ((\Box a) \to (\Box b))} \arrow[d, "R\to"] & \\
& {(\Box a), (\Box (a \to b)) => (\Box b)} \arrow[d, "K\Box"] & \\
& {a, (a \to b) => b} \arrow[dl, "L\to"] \arrow[dr, "L\to"] & \\
{a => a, b} \arrow[d, "Ax"] & & {a, b => b} \arrow[d, "Ax"] \\
\Box & & \Box
\end{tikzcd}$$

6 Test and Benchmarks

Given these provers, we now want to test them for correctness and benchmark their performance. In this chapter, we will explain the formulas we use for tests and benchmarks

6.1 Formulas

6.1.1 Unit Tests

In table 1, we list the formulas we use to test the correctness of our provers. We include common examples and ensure that we have both positive and negative examples for each logic.

Table 1: Provabilty of Formulas

	G3C	G3I	G3K
$\bot$	×	×	×
$\top$	✓	✓	✓
$p \wedge \neg p$	×	×	×
$p \to p$	✓	✓	✓
$r \wedge (p \vee \neg p)$	×	×	×
Double negation: $p \leftrightarrow \neg\neg p$	✓	×	✓
Excluded middle: $p \vee \neg p$	✓	×	✓
Pierce's Law $((p \to q) \to p) \to\ p$	✓	×	✓
$\neg\neg(p \vee \neg p)$	✓	✓	✓
$p \to \neg\neg p$	✓	✓	✓
$(p \to (p \to q)) \to (p \to q)$	✓	✓	✓
`phi`: $(p \wedge (p \to q)) \to ((p \to q) \to q)$	✓	✓	✓
$((((p \to q) \to p) \to\ p) \to q) \to q$	✓	✓	✓
$\Box\top$	-	-	✓
$\Box\bot$	-	-	×
K axiom: $\Box(p \to q) \to (\Box p \to \Box q)$	-	-	✓
$\Box(p \to q) \to (\Box p \to (\Box q \to \Box r))$	-	-	×

6.1.2 Parameterized Formulas

We also define functions that take integers as input to generate formulas of increasing size. We call the resulting formulas parameterized formulas and will mainly use them for benchmarks.

For some formulas, we also distinguish them by the function we use to generate them, `foldr` and `foldl`. For more information about these two functions, please see [Hut16]. The functions that end with "R" are generated by `foldr`, "L" otherwise.

To illustrate the distinction between them, we will use the examples of `disPhiPieR` and `disPhiPieL`. In this example, the formula `phi` is the formula $(p \wedge (p \to q)) \to ((p \to q) \to q)$ in

Table 1, valid in intuitionistic propositional logic.

```
disPhiPieR :: Int -> FormP
disPhiPieR k = foldr DisP phi (replicate (2*k) pierce )

disPhiPieL :: Int -> FormP
disPhiPieL k = foldl DisP phi (replicate (2*k) pierce )
```

We show the syntax trees for `disPhiPieR 2` and `disPhiPieL 2` in Figure 1 and Figure 2.

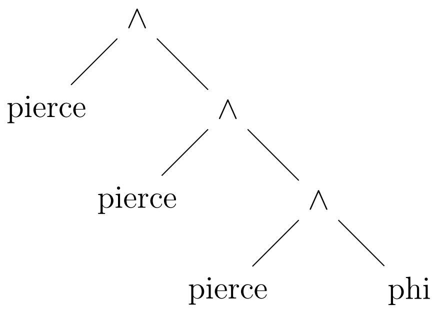


Figure 1: disPhiPieR 2

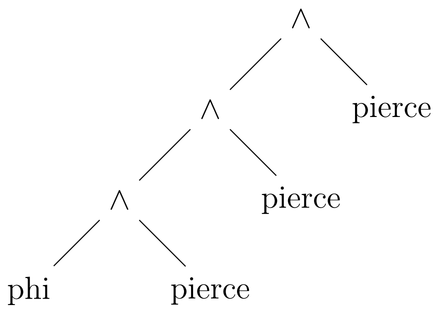


Figure 2: disPhiPieL 2

For arbitrary n, `disPhiPieR n` is logically equivalent to `disPhiPieL n` in any logic. But as seen in Figure 1 and Figure 2, their constructions are not the same. Due to our prover for the zipper being left-biased, we want to know if there will be a difference (whether in time or in memory use) between them.

The rest of the parameterized functions are:

```
conBotR :: Int -> FormP
conBotR k = foldr ConP BotP (replicate k BotP )

-- ghci> conBotR 2
-- (⊥ ∧ (⊥ ∧ ⊥))

conBotL :: Int -> FormP
conBotL k = foldl ConP BotP (replicate k BotP )

-- ghci> conBotL 2
-- ((⊥ ∧ ⊥) ∧ ⊥)

conPieR :: Int -> FormP
conPieR k = foldr ConP pierce (replicate (2*k) pierce )

-- ghci> conPieR 2
-- ((((p → q) → p) → p) ∧ ((((p → q) → p) → p) ∧ ((((p → q) → p) → p) ∧ ((((p → q) → p) → p
    ) ∧ (((p → q) → p) → p)))))

conPieL :: Int -> FormP
conPieL k = foldl ConP pierce (replicate (2*k) pierce )

-- ghci> conPieL 2
-- (((((((p → q) → p) → p) ∧ (((p → q) → p) → p)) ∧ (((p → q) → p) → p)) ∧ (((p → q) → p) →
     p)) ∧ (((p → q) → p) → p))

disPieR :: Int -> FormP
disPieR k = foldr DisP pierce (replicate (2*k) pierce )

-- ghci> disPieR 2
-- ((((p → q) → p) → p) v ((((p → q) → p) → p) v ((((p → q) → p) → p) v ((((p → q) → p) → p
    ) v (((p → q) → p) → p)))))

disPieL :: Int -> FormP
disPieL k = foldl DisP pierce (replicate (2*k) pierce )
```

```
-- ghci> disPieL 2
-- (((((((p → q) → p) → p) v (((p → q) → p) → p)) v (((p → q) → p) → p)) v (((p → q) → p) →
     p)) v (((p → q) → p) → p))
```

We also add some extra tests for our modal logic K.

```
boxesTop :: Int -> FormM
boxesTop 0 = topM
boxesTop n = Box (boxesTop (n-1))

--ghci> boxesTop 3
--( □ ( □ ( □ (⊥ → ⊥))))

boxesBot :: Int -> FormM
boxesBot 0 = BotM
boxesBot n = Box (boxesBot(n-1))

--ghci> boxesBot 3
--( □ ( □ ( □ ⊥)))

listOfAt :: Int -> [FormM]
listOfAt n = map AtM $ take n ['c'..]

formForK :: Int -> FormM
formForK n = ImpM (Box (List.foldr ImpM (AtM 'a') (listOfAt n)))
                $ foldr (ImpM . Box) (Box (AtM 'a')) (listOfAt n)
--ghci> formForK 3
--(( □ (c → (d → (e → a)))) → (( □ c) → (( □ d) → (( □ e) → ( □ a)))))

nFormForK :: Int -> FormM
nFormForK n = ImpM (Box (List.foldr ImpM (AtM 'a') (listOfAt n ++ [AtM 'b'])))
                $ foldr (ImpM . Box) (Box (AtM 'a')) (listOfAt n)

--ghci> nFormForK 3
--(( □ (c → (d → (e → (b → a))))) → (( □ c) → (( □ d) → (( □ e) → ( □ a)))))
```

Here, `formForK n` generates variants of the K axiom with $n + 1$ propositional variables, and `nFormForK n` adds an extra variable b in the antecedent. Thus, `formForK n` is provable whereas `nFormForK n` is not provable. We show the provability in Table 2 for these parameterized formulas:

Table 2: Provability of Parametrized Formulas

Parametrized Formulas	G3C	G3I	G3K
conBotR(L)	×	×	×
conPieR(L)	✓	×	✓
disPieR(L)	✓	×	✓
disPhiPieR(L)	✓	✓	✓
boxesTop	-	-	✓
boxesBot	-	-	×
formForK	-	-	✓
nFormForK	-	-	×

6.2 Additional Correctness Tests

Besides the tests in Section 6.1, we also want to run integration tests. That is, we want to check that certain conditions are always satisfied regardless of the input formulas to our provers. This is doable since we have defined `Arbitrary` instances for our formulas. We check the following:

- The results of the provability of a given formula should be identical for both the tree-based and zipper-based provers.
- The proof trees obtained by each prover for any given formula must be at most binary.
- In any logic, if f and g are provable, then their conjunction should also be provable.
- If a formula is provable in G3C, then its double negation should be provable in G3I.
- If a formula is provable in G3C, then it should also be provable in G3K.

7 Results

This chapter presents the results for test, run time and memory usage separately.

7.1 Test Results

All tests described in Section 6.1 pass and demonstrate the correctness of the implemented provers. Below is the result after running `stack test`.

```
Unit tests
G3C.isProvableZ
  Top [✓]
  (p → p) [✓]
  Double negation: ((((p → ⊥) → ⊥) → p) ∧ (p → ((p → ⊥) → ⊥))
     ) [✓]
  Excluded middle: (p v (p → ⊥)) [✓]
  Pierce's law: (((p → q) → p) → p) [✓]
  (((p v (p → ⊥)) → ⊥) → ⊥) [✓]
  (p → ((p → ⊥) → ⊥)) [✓]
  ((p → (p → q)) → (p → q)) [✓]
  ((p ∧ (p → q)) → ((p → q) → q)) [✓]
  (((((p → q) → p) → p) → q) → q) [✓]
not.G3C.isProvableZ
  Bot [✓]
  (p ∧ (p → ⊥))  [✓]
  (r ∧ (p v (p → ⊥)))  [✓]
G3C.isProvableT
  Top [✓]
  (p → p) [✓]
  Double negation: ((((p → ⊥) → ⊥) → p) ∧ (p → ((p → ⊥) → ⊥))
     ) [✓]
  Excluded middle: (p v (p → ⊥)) [✓]
  Pierce's law: (((p → q) → p) → p) [✓]
```

```
    (((p v (p → ⊥)) → ⊥) → ⊥) [✓]
    (p → ((p → ⊥) → ⊥)) [✓]
    ((p → (p → q)) → (p → q)) [✓]
    ((p ∧ (p → q)) → ((p → q) → q)) [✓]
    (((((p → q) → p) → p) → q) → q) [✓]
  not.G3C.isProvableT
    Bot [✓]
    (p ∧ (p → ⊥))  [✓]
    (r ∧ (p v (p → ⊥)))  [✓]
  G3I.isProvableZ
    Top [✓]
    (p → p) [✓]
    (((p v (p → ⊥)) → ⊥) → ⊥) [✓]
    (p → ((p → ⊥) → ⊥)) [✓]
    ((p → (p → q)) → (p → q)) [✓]
    ((p ∧ (p → q)) → ((p → q) → q)) [✓]
    (((((p → q) → p) → p) → q) → q) [✓]
  not.G3I.isProvableZ
    Bot [✓]
    (p ∧ (p → ⊥))  [✓]
    (r ∧ (p v (p → ⊥)))  [✓]
    Double negation: ((((p → ⊥) → ⊥) → p) ∧ (p → ((p → ⊥) → ⊥))
      ) [✓]
    Excluded middle: (p v (p → ⊥)) [✓]
    Pierce's law: (((p → q) → p) → p) [✓]
  G3I.isProvableT
    Top [✓]
    (p → p) [✓]
    (((p v (p → ⊥)) → ⊥) → ⊥) [✓]
    (p → ((p → ⊥) → ⊥)) [✓]
    ((p → (p → q)) → (p → q)) [✓]
    ((p ∧ (p → q)) → ((p → q) → q)) [✓]
    (((((p → q) → p) → p) → q) → q) [✓]
  not.G3I.isProvableT
    Bot [✓]
    (p ∧ (p → ⊥))  [✓]
    (r ∧ (p v (p → ⊥)))  [✓]
    Double negation: ((((p → ⊥) → ⊥) → p) ∧ (p → ((p → ⊥) → ⊥))
      ) [✓]
    Excluded middle: (p v (p → ⊥)) [✓]
    Pierce's law: (((p → q) → p) → p) [✓]
  G3K.isProvableZ
    Top [✓]
    (p → p) [✓]
    Double negation: ((((p → ⊥) → ⊥) → p) ∧ (p → ((p → ⊥) → ⊥))
      ) [✓]
    Excluded middle: (p v (p → ⊥)) [✓]
    Pierce's law: (((p → q) → p) → p) [✓]
    (((p v (p → ⊥)) → ⊥) → ⊥) [✓]
    (p → ((p → ⊥) → ⊥)) [✓]
    ((p → (p → q)) → (p → q)) [✓]
    ((p ∧ (p → q)) → ((p → q) → q)) [✓]
    (((((p → q) → p) → p) → q) → q) [✓]
    Box top [✓]
    K axiom [✓]
  not.G3K.isProvableZ
    Bot [✓]
    (p ∧ (p → ⊥))  [✓]
    (r ∧ (p v (p → ⊥)))  [✓]
    Box ⊥ [✓]
    (( □ (a → b)) → (( □ a) → (( □ b) → ( □ c)))) [✓]
  G3K.isProvableT
    Top [✓]
    (p → p) [✓]
    Double negation: ((((p → ⊥) → ⊥) → p) ∧ (p → ((p → ⊥) → ⊥))
      ) [✓]
    Excluded middle: (p v (p → ⊥)) [✓]
    Pierce's law: (((p → q) → p) → p) [✓]
    (((p v (p → ⊥)) → ⊥) → ⊥) [✓]
    (p → ((p → ⊥) → ⊥)) [✓]
    ((p → (p → q)) → (p → q)) [✓]
    ((p ∧ (p → q)) → ((p → q) → q)) [✓]
    (((((p → q) → p) → p) → q) → q) [✓]
    Box top [✓]
    K axiom [✓]
  not.G3K.isProvableT
    Bot [✓]
    (p ∧ (p → ⊥))  [✓]
    (r ∧ (p v (p → ⊥)))  [✓]
    Box ⊥ [✓]
    (( □ (a → b)) → (( □ a) → (( □ b) → ( □ c)))) [✓]
Additional Correctness Tests
  Equivalence between two provers
    In G3C [✓]
      +++ OK, passed 1000 tests.
    In G3I [✓]
      +++ OK, passed 1000 tests.
    In G3K [✓]
      +++ OK, passed 1000 tests.
  Proofs are at most binary
    zipper for G3C [✓]
      +++ OK, passed 100 tests.
    tree for G3C [✓]
      +++ OK, passed 100 tests.
    zipper for G3I [✓]
      +++ OK, passed 100 tests.
    tree for G3I [✓]
      +++ OK, passed 100 tests.
    zipper for G3K [✓]
      +++ OK, passed 100 tests.
    tree for G3K [✓]
      +++ OK, passed 100 tests.
  If f and g isProvable, then Con f g isProvable
    zipper for G3C [✓]
      +++ OK, passed 100 tests.
    tree for G3C [✓]
      +++ OK, passed 100 tests.
    zipper for G3I [✓]
      +++ OK, passed 100 tests.
    tree for G3I [✓]
      +++ OK, passed 100 tests.
    zipper for G3K [✓]
      +++ OK, passed 100 tests.
    tree for G3K [✓]
      +++ OK, passed 100 tests.
  If f isProvable in G3C, then neg neg f isProvable in G3I
    zipper [✓]
      +++ OK, passed 100 tests.
    tree [✓]
      +++ OK, passed 100 tests.
  If f isProvable in G3C, then f isProvable in G3K
    zipper [✓]
      +++ OK, passed 100 tests.
    tree [✓]
      +++ OK, passed 100 tests.

Finished in 0.0460 seconds
105 examples, 0 failures

As we can see, all the test conditions outlined in Chapter 6
   are passed.
```

7.2 Run Time

We use the Criterion Haskell library [O'S24] to obtain benchmark results for run time. Given that we have many parameterized formulas to test, and many of them are very similar, here we only present the run time results of the most interesting cases. All other cases are pretty similar to ConPie for G3I.

- conBot for G3C, G3I, and G3K.
- conPie for G3I.

- `boxesTop` for G3K.

These cases highlight the prominent differences between the two provers. For additional run time results, we refer to the "Plot" folder in the same route.

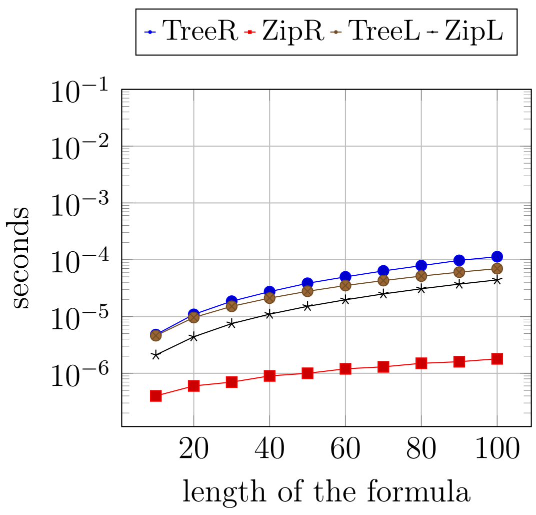


Figure 3: G3C conBot

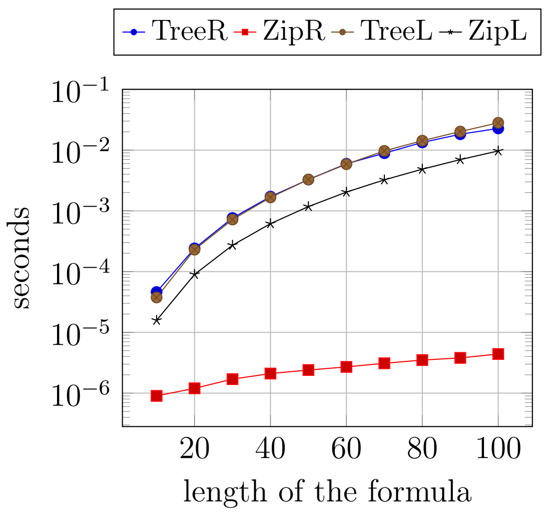


Figure 4: G3I conBot

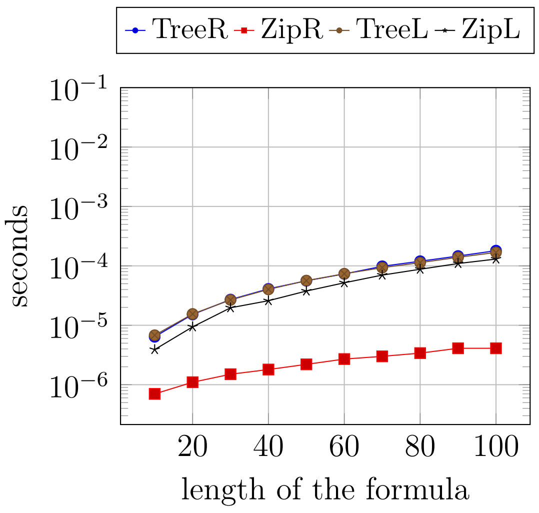


Figure 5: G3K conBot

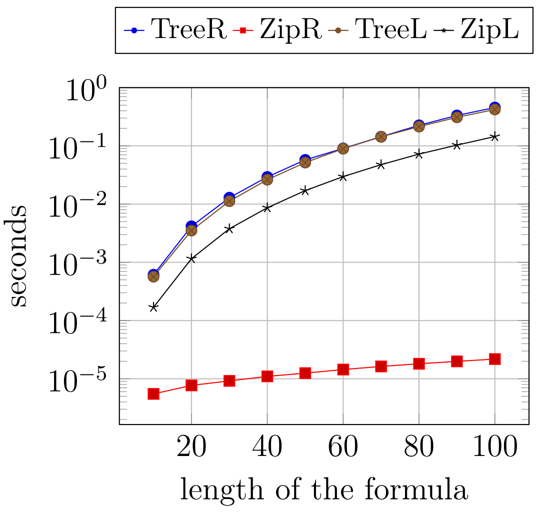


Figure 6: G3I conPie
(note the different y-axis)

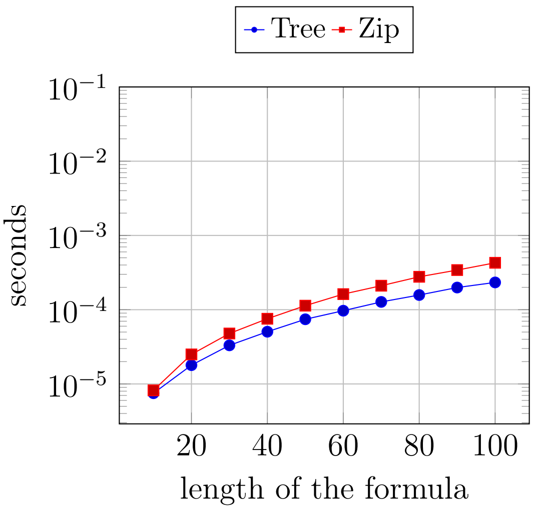


Figure 7: G3K boxesTop

Figures 3, 4, and 5 each represent the comparison between the zipper-based prover and the tree-based prover for the formula `conBot` in the proof systems G3C, G3I, and G3K, respectively. Figure 6 shows the comparison for the formula `conPie` in G3I.

In these four cases, each graph contains four lines. Lines starting with "Zip" represent the runtime of the zipper-based prover, and lines starting with "Tree" represent the runtime of the tree-based prover. Lines ending with "L" indicate that the formula was constructed using `foldl`, while lines ending with "R" indicate construction using `foldr`. For example, the line "ZipR" in Figure 3 represents the runtime of the zipper-based prover trying to prove `conBotR n` as n grows from 0 to 100.

In all these cases, the formulas are not provable. As we can see, in all cases, the zipper-based prover is always faster than the tree-based prover, and formulas constructed using `foldr` are dealt-with faster than those constructed using `foldl`. Additionally, in the case of `foldr`, the speed advantage of the zipper-based prover is significantly greater than in the case of `foldl`.

Figure 7 is different, as `boxesTop` is not constructed based on `foldr` or `foldl`. Therefore, it only has two lines: one for the zipper-based prover and one for the tree-based prover. It is also the only case, including those not shown here, where the zipper-based prover becomes increasingly slower than the tree-based prover as the formula's length grows.

7.3 Memory Usage

We use the Weigh library [Don23] to measure the memory usage when proving parameterized formulas. In Table 3:

- "Allocated" refers to the amount of memory allocated during execution. This value is measured in bytes and indicates the amount of memory the program has requested from the system for its operations.

- "GCs" is an abbreviation for Garbage Collection. Garbage collection reclaims memory that a program no longer needs, preventing memory leaks. In these tests, "GCs" refers to the number of times the program performed garbage collection. Fewer garbage collections typically indicate better memory management and performance.

Table 3: Memory Usage Benchmarks

Logic	Formula	Type	Size	Result	Allocated	GCs
G3C	disPhiPieR	Tree	100	True	445,744	0
G3C	disPhiPieR	Zip	100	True	337,632	0
G3C	disPhiPieL	Tree	100	True	440,952	0
G3C	disPhiPieL	Zip	100	True	332,840	0
G3C	disPieR	Tree	100	True	440,616	0
G3C	disPieR	Zip	100	True	333,032	0
G3C	disPieL	Tree	100	True	435,824	0
G3C	disPieL	Zip	100	True	328,240	0
G3C	conPieR	Tree	100	True	1,836,256	0
G3C	conPieR	Zip	100	True	1,290,272	0
G3C	conPieL	Tree	100	True	1,831,504	0
G3C	conPieL	Zip	100	True	1,285,520	0
G3C	conBotR	Tree	100	False	229,440	0
G3C	conBotR	Zip	100	False	6,304	0
G3C	conBotL	Tree	100	False	246,888	0
G3C	conBotL	Zip	100	False	118,792	0
G3I	disPhiPieR	Tree	100	True	13,108,920	3
G3I	disPhiPieR	Zip	100	True	13,092,920	3
G3I	disPhiPieL	Tree	100	True	14,482,200	3
G3I	disPhiPieL	Zip	100	True	14,466,192	3
G3I	disPieR	Tree	100	False	12,888,976	3
G3I	disPieR	Zip	100	False	12,748,704	3
G3I	disPieL	Tree	100	False	14,235,424	3
G3I	disPieL	Zip	100	False	14,095,152	3
G3I	conPieR	Tree	100	False	33,065,344	8
G3I	conPieR	Zip	100	False	24,648	0
G3I	conPieL	Tree	100	False	33,067,624	8
G3I	conPieL	Zip	100	False	4,792,176	1
G3I	conBotR	Tree	100	False	3,494,104	0
G3I	conBotR	Zip	100	False	7,088	0
G3I	conBotL	Tree	100	False	3,511,552	0
G3I	conBotL	Zip	100	False	1,250,736	0
G3M	disPhiPieR	Tree	100	True	516,600	0
G3M	disPhiPieR	Zip	100	True	408,488	0
G3M	disPhiPieL	Tree	100	True	511,808	0
G3M	disPhiPieL	Zip	100	True	403,696	0
G3M	disPieR	Tree	100	True	511,296	0
G3M	disPieR	Zip	100	True	403,712	0
G3M	disPieL	Tree	100	True	506,504	0

Continued on next page

Table 3: Performance Benchmarks (continued)

Logic	Formula	Type	Size	Result	Allocated	GCs
G3M	disPieL	Zip	100	True	398,920	0
G3M	conPieR	Tree	100	True	1,906,936	0
G3M	conPieR	Zip	100	True	1,360,952	0
G3M	conPieL	Tree	100	True	1,902,184	0
G3M	conPieL	Zip	100	True	1,356,200	0
G3M	conBotR	Tree	100	False	239,064	0
G3M	conBotR	Zip	100	False	13,528	0
G3M	conBotL	Tree	100	False	256,512	0
G3M	conBotL	Zip	100	False	126,016	0
G3M	boxesTop	Tree	1000	True	76,662,600	13
G3M	boxesTop	Zip	1000	True	272,224,024	54
G3M	boxesBot	Tree	1000	False	1,818,048	0
G3M	boxesBot	Zip	1000	False	1,505,184	0
G3M	formForK	Tree	20	True	85,799,671,352	20,518
G3M	formForK	Zip	20	True	85,799,650,696	20,518
G3M	nFormForK	Tree	20	False	85,799,677,072	20,518
G3M	nFormForK	Zip	20	False	85,799,564,808	20,518

In most cases, memory usage is quite similar between the provers. The zipper-based prover generally uses fewer bytes in most cases. However, in some cases (colored red in the table), the prominence of the zipper's lower memory usage (allocated) is more obvious. These cases are `conBot` in G3C, G3I, and G3K, and `conPie` in G3I.

Another phenomenon observed is that in these cases, the advantage of the zipper is more apparent for formulas constructed by `foldr` than those constructed by `foldl`. For example, in the case of G3I `conPieR`, the memory used by the tree-based prover is 1342 times that of the zipper-based prover. However, for G3I `conPieL`, the memory used by the tree-based prover is only 7 times that of the zipper-based prover.

The ratio comparison of these cases is shown in Table 4. The ratio is the ratio of the allocated memory of the tree prover to the allocated memory of the zipper prover, rounded to the nearest integer.

Table 4: The Ratio Comparisons

Logic	Formula	Foldr Ratio $[\frac{\text{Tree allocated}}{\text{Zipper allocated}}]$	Foldl Ratio $[\frac{\text{Tree allocated}}{\text{Zipper allocated}}]$
G3C	conBot	36	2
G3I	conPie	1342	7
G3I	conBot	493	3
G3K	conBot	18	2

Also, in the case of `conPie`, the tree-based prover performs garbage collection 8 times, whereas the zipper-based prover does it 0 or 1 time. However, in the case of `boxes_top`, the zipper-based prover uses almost 3 times the memory of the tree-based prover and does more garbage

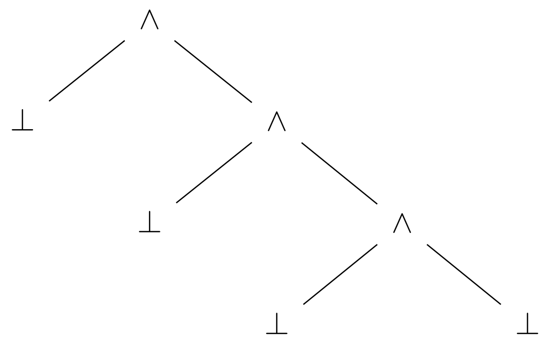


Figure 8: conBotR 3

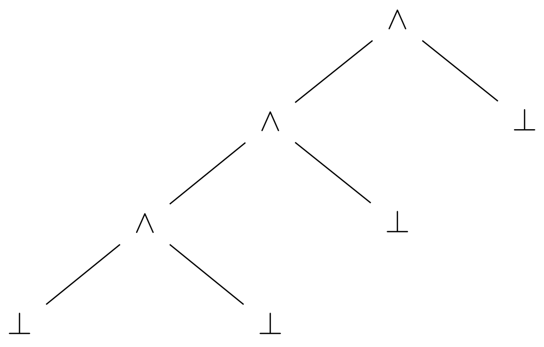


Figure 9: conBotL 3

collections.

7.4 Discussion

The provers' time and memory usage are generally quite close. The zipper-based prover generally uses fewer bytes and less time.

The formula `boxes_top` is a special case (and the only such case in our test set) where the zipper-based prover is significantly slower and uses more memory than the tree-based prover. Unfortunately, we do not know the reason behind this.

In four cases that we frequently mentioned: `conBot` in G3C, G3I, and G3K, as well as `conPie` in G3I, the advantage of shorter run time and lower memory usage of the zipper-based prover, is much more prominent than in other cases. This is particularly noticeable when the formula is constructed by `foldr`.

Another observation is that these are all cases where the formula is not provable. We claim that this difference is due to the ease of navigating and modifying the zipper prover, which stops immediately upon encountering an open branch. In contrast, the tree-based prover continues to extend the other branches regardless. So, the zipper-based prover has fewer needs to perform garbage collection, especially in the case where the formula is more complicated, such as `conPie`. This feature is then reinforced by the structural differences between `foldl` and `foldr` because the principle of extending the proof for the zipper is left-biased.

To understand this result, let's first consider the structure of formulas generated by `foldl` and `foldr`. Similar to Figure 2 and Figure 1, we now present the syntax trees for `conBotR 3` and `conBotL 3`.

Case conBotR As shown in Figure 8, we notice that $\bot$ appears as the highest left branch of `conBotR`. This means that after the first rule R$\wedge$ is applied, we will already have a left $\bot$ in our next sequent.

Our prover for the zipper is left-biased, meaning it will move directly to this branch: a single node containing $\bot$ before proving the other branch. Once the prover realizes that there is nothing that can be done for this branch, it will return: `Node (singleton (Right BotP)) ‘“’ []` for this branch and instantly stop the whole proving process.

Whereas for the tree, it will create two sub-trees: the left one is a single node containing $\bot$, and the right one is a single node containing $\bot \wedge (\bot \wedge \bot)$. It will extend both trees at the same time. That is, the end of the first branch will not impact the second one. So even if the first branch is a dead end, it will still extend the second branch.

Case conBotL However, in Figure 9, $\bot$ is the highest right branch of `conBotL`. So after the first rule R$\wedge$ is applied, we will have two sub-trees: the left one is a single node containing $\bot \wedge (\bot \wedge \bot)$, and the right one is a single node containing $\bot$.

Our prover for the zipper will first try to extend the left branch, so it will keep applying R$\wedge$ two times and focusing on extending the left subtree until getting a single node containing $\bot$. Then it will stop the proving process.

Whereas for the tree, it will also apply R$\wedge$ while, in the meantime, also trying to extend the branches created in the process.

Summary To summarize, the zipper-based prover is much more efficient in these cases due to its ability to focus on promising parts of the proof tree and quickly discard irrelevant branches, especially with `foldr`-constructed formulas.

When encountering a branching rule, the zipper-based prover prioritizes the left branch, quickly stopping further exploration when encountering a dead end, while the tree-based prover explores all branches simultaneously. `foldr`-constructed formulas align well with the zipper prover, placing critical parts early and allowing for faster identification of dead ends.

The differences are also reflected in the frequency of garbage collection. The zipper-based prover requires fewer garbage collections, indicating better memory management. However, exceptions like `boxes_top` show the zipper-based approach using more memory, suggesting its performance depends on formula structure. This highlights the necessity for further optimisation and a deeper understanding of the zipper-based approach.

8 Future Work

Several directions could be explored in future work.

1. Firstly, with our two generic modular provers, we can extend our implementation to other logics, such as T, S4, and S5, in modal logic. These logics are commonly used in various fields, such as computer science and philosophy. Implementing these logics should be easy. We juse need to add more unsafe rules for each logic and modify our provers to handle the case where there might be more than one unsafe rule.

2. Secondly, testing more complex cases in more complex logic systems could further optimize our zipper-based prover. Although we already have many test cases, increasing their complexity and variety will help identify more patterns and discover potential areas for improvement.

3. Furthermore, the zipper-based prover's prominent efficiency in identifying non-provable formulas indicates the potential for employing the zipper structure in proof systems

focused on finding contradictions, such as the tableau method. Its main goal is to find contradictions to determine the validity of a given formula. By using the zipper for the tableau system, we can quickly traverse and discard redundant branches to find a contradiction more efficiently.

9 Acknowledgement

I would like to express my deepest gratitude to my supervisor, Dr. Malvin Gattinger, for his invaluable guidance, support, and encouragement throughout my research. His insights and expertise have been instrumental in shaping this thesis.

I am also grateful to my committee members, Dr. Benno van den Berg, Dr. Andres Goens Jokisch, and Dr. Marianna Girlando, for their valuable feedback. Special thanks to Dr. Marianna Girlando for providing many suggestions and guidance during my research and introducing propositional intuitionistic logic to enrich our proving patterns.

I would like to thank my friends at ILLC for their support, discussions, and encouragement during this journey. Their insights have made this experience truly enriching: Alyssia Renata, Chu TianYi, Hu Ruiting, Justus Becker, Liam Chung, Raufs Dunamalijevs, Vince Chang, Wang Haitian, and Ye Lingyuan.

To my boyfriend Wang Yilun, thank you for your unwavering support in both my studies and daily life.

To my parents Yang Liu and Yang Yi, thank you for your endless patience and belief in me.

I would also like to thank the CrossFit Amsterdam community, including coaches Alex, Clay, and Nick, and my friends Dionne, Esther, Hayley (and her cat Poki), and everyone who has cheered me on. One year of CrossFit has truly brightened my mind and given me countless joyful memories.

I would like to thank my undergraduate friends Zhang Haoyu and Ji Yu for staying connected across the distance.

To my therapist Amy and psychiatrist Alain, thank you for helping me regulate my emotions and believe in myself.

Thank you all.

References

[BDRV01] Patrick Blackburn, Maarten De Rijke, and Yde Venema. *Modal Logic*, volume 53. Cambridge University Press, 2001.

[Don23] Chris Done. Weigh, 2023. Haskell library. URL: `https://github.com/fpco/weigh`.

[Dyc16] Roy Dyckhoff. Intuitionistic Decision Procedures since Gentzen. *Advances in proof theory*, pages 245–267, 2016.

[Gat23a] Malvin Gattinger. FP2023 lecture 9: Zippers and the Algebra of Data Types. Unpublished lecture slides, 2023.

[Gat23b] Malvin Gattinger. Modal tableau interpolation, 2023. URL: `https://github.com/m4lvin/modal-tableau-interpolation`.

[Gat24] Malvin Gattinger. Smcdel, 2024. URL: `https://github.com/jrclogic/SMCDEL`.

[GHLS19] Olivier Gasquet, Andreas Herzig, Dominique Longin, and Mohamad Sahade. Lotrec, 2019. URL: `https://www.irit.fr/Lotrec/`.

[Hue97] Gérard Huet. The Zipper. *Journal of functional programming*, 7(5):549–554, 1997.

[Hut16] Graham Hutton. *Programming in Haskell*, chapter 7, pages 73–91. Cambridge University Press, 2016.

[Mae54] Shôji Maehara. Eine Darstellung der Intuitionistischen Logik in der Klassischen. *Nagoya Mathematical Journal*, 7:45–64, 1954. `doi:10.1017/S0027763000018055`.

[MS23] Ivan Lazar Miljenovic Matthew Sackman. Graphviz, 2023. Haskell library. URL: `https://gitlab.com/daniel-casanueva/haskell/graphviz`.

[O'S24] Bryan O'Sullivan. Criterion, 2024. Haskell library. URL: `https://github.com/haskell/criterion`.

[TS00] Anne Sjerp Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 2000.