GenZ: A Generic Sequent Calculus Prover using the Zipper

Xiaoshuang Yang[1], Malvin Gattinger[2], Marianna Girlando[2,3] and Patrick Koopmann[1]

[1]Vrije Universiteit Amsterdam, De Boelelaan 1105, 1081 HV Amsterdam, The Netherlands
[2]University of Amsterdam, Spui 21, 1012 WX Amsterdam, The Netherlands
[3]University of Southern Denmark, Campusvej 55 5230 Odense, Denmark

Abstract

We introduce GenZ, a generic theorem prover for sequent calculi implemented in Haskell. Sequent calculus is a simple and versatile formalism, widely used to define proof systems for modal and non-classical logics. GenZ allows the user to specify a set of sequent rules, over which it performs proof search. This makes it possible to rapidly implement and test proof systems for a wide variety of logics, a useful feature for both research and teaching. To allow for efficient proof search, GenZ employs the zipper data structure. We illustrate our system by implementing 11 well-known sequent calculi for classical and intuitionistic propositional logics, as well as for several modal logics from the S5 cube, and evaluate them on formulas from the LWB and ILTP benchmark suites.

Keywords

Theorem Proving, Sequent Calculus, Zipper Data Structure

1. Introduction

Since their introduction by Gentzen in 1935 [1], proof systems based on *sequents* have been defined for a wide variety of non-classical logics. The sequent calculus usually allows for syntactically simulating an explicit structural rule to compose derivations: the *cut rule*. Instead, Hilbert-style axiom systems usually rely on *modus ponens*, which is similar to the cut rule. Admissibility of cut is thus helpful in showing completeness of sequent calculi with respect to axiom systems. If the cut rule is admissible in a proof system (that is: if it can be simulated by other rules of the calculus), the proof system often turns out to be *analytic*, meaning that all the formulas occurring in a cut-free proof are subformulas of the formula at the root of the derivation. As a consequence, sequent calculus (together with other cut-free formalisms, such as tableaux), is suitable to implement an automated root-first proof search for a formula. That is, one can define a *decision algorithm* which tests the validity of a formula by trying to construct (bottom-up) a sequent calculus proof for it. Compared to tableaux, sequent calculus strikes as an elegant and simple system: in particular, it is easy to identify the reasoning steps in a sequent calculus proof, facilitating explanability. The role of proofs in explaining reasoning is now recognised [2, 3, 4], as showcased by systems like Evonne that specialize in visualizing proofs to end-users [5].

A *theorem prover* is a tool implementing proof-theoretic decision algorithms. While there exist several theorem provers based on sequent calculus, these are highly *specialised*, meaning that they target specific logics, aiming at optimal performance. Such sequent-based theorem provers include: a history-based theorem prover IntHistGC for intuitionistic logic [6] based on [7], the NESCOND prover for conditional logics [8], and the MOIN prover [9] for nested sequents [10].

In contrast to the specialised approach, the quest for *generic* theorem provers has gained considerable attention. Generic provers are particularly helpful for research and teaching purposes, as they allow to implement the rules of *any* proof system, and quickly test them out. For instance, a researcher could use this tool to test if the proof system they have implemented derives all the axioms, or terminates.

Practical Aspects of Automated Reasoning (PAAR) 2026, as part of the Federated Logic Conference (FLoC) 2026, Lisbon, Portugal, July 25, 2026

x.s.yang@vu.nl (X. Yang); malvin@w4eg.eu (M. Gattinger); p.k.koopmann@vu.nl (P. Koopmann)
https://malv.in/ (M. Gattinger); https://www.mariannagirlando.com (M. Girlando); https://pkoopmann.github.io/ (P. Koopmann)
0009-0009-0735-3822 (X. Yang); 0000-0002-2498-5073 (M. Gattinger); 0000-0002-9384-1356 (M. Girlando); 0000-0001-5999-2583 (P. Koopmann)

published 2026-08-20

Similarly, a student could use a generic prover to understand how the construction of proofs works in a specific system.

There are several ways in which a prover can be qualified as 'generic'. Some systems focus on automating the proofs of *meta-properties* of a sequent calculus entered by the user (e.g., cut admissibility). To this approach belong provers such as Quati [11] and the L-Framework [12], which possibly has the highest degree of generality. The definition of algorithms that generate a proof-search strategy from a set of sequent rules entered by the user has also been explored, in [13]. With a different aim, there exist several generic provers that are actually about proving theorems using the rules of the proof system, and are designed to be adaptable to a wide range of calculi, sometimes specified by the user. Such implementations include provers based on tableaux calculi, such as LoTREC [14], JTabWb [15] and MetTeL, MetTeL^2 [16, 17], the nanoCoP provers [18] for classical, intuitionistic and modal logics (Prolog), and the Cyclist prover [19] for FOL and separation logic (OCaml). For sequent calculus, we mention the generic provers Gen2sat [20] and Sequoia [21].

We here introduce GenZ, a generic theorem prover based on sequent calculus, written in Haskell. The prover is 'generic' in the second sense described above: users specify the rules of their favourite sequent calculus (by means of simple Haskell coding), and GenZ performs proof search using these rules. Differently from the other generic provers in the literature, GenZ focuses on producing terminating proof search trees: the user can select some (basic) termination strategies currently implemented by the prover. Thus, GenZ does not aim at meta-theory and is meant to be a lightweight tool, allowing the user to specify any set of rules (not necessarily sound) and perform proof search on them.

We implemented a total of 11 cut-free sequent calculus systems in GenZ: for classical and intuitionistic propositional logic, for eight modal logics in the S5-cube and for Gödel-Löb logic (GL), for which we chose a cyclic proof system. While GenZ does not aim to compete with specialised provers for these logics, it is a tool suitable for both research and teaching, as it allows one to construct and visualise sequent calculus proofs automatically. The source code is available at https://github.com/XiaoshuangYang999/GenZ and the prover can be used through a web interface at https://tools.malv.in/genz-web.

The proof search is implemented in GenZ using the *zipper*, a family of data structures introduced by Huet in [22], allowing for efficient navigation of hierarchical data structures, such as trees. The zipper was also used in the Haskell prover MOLTAP [23], implementing tableaux for modal logics. To understand the impact of the zipper on the performance, we also implemented a more direct version of the proof search that uses a tree data structure, and compared the two on a benchmark of generated formulas, showing that the zipper indeed leads to substantial reductions in memory usage. Our evaluation also includes formulas from known benchmarks, namely the *Logics Workbench* (LWB) [24] and the ILTP library [25], showing that GenZ can often (dis-)prove formulas of tree size below 200 in seconds.

Comparison with the literature. Compared with Gen2sat [20] and Sequoia [21], sequent calculus provers which allow the user to specify rules of their choice, GenZ is more general. Namely, both Gen2sat and Sequoia only support so-called *pure* rules, which are rules that propagate the full context (the non-principal formulas) from the conclusion to every premise of the rule. Rules needed for modal logics, such as the modal rule $\Box_k$ (refer to Section 2), are not pure, whence sequent calculi for modal logics cannot be implemented in these provers, or at least not in their usual form. By contrast, sequent calculi for modal logics (and for many other proof systems) can be implemented in GenZ. Moreover, Gen2sat reduces reasoning to SAT and does not produce proofs, which is a crucial feature of GenZ, and Sequoia does not produce proofs automatically, but only assists users in generating proofs manually. GenZ instead is about the automatic generation of proofs: it searches for proofs autonomously, applies loop checks, and displays complete proof trees as well as failed attempts, allowing users to visualise and reason about the results.

The L-Framework [12] allows specifying generic rules, constructing proofs in them, and proving properties of the proof system. The goals of the L-Framework are more general than the goals of GenZ, but the system is limited in another aspect. GenZ adds to the proof search genuine termination strategies, reflecting what is done in the proof theoretic literature. In contrast, termination of proof

search is not explicitly discussed in [12], as proof search is handled by a rewriting system.

Structure of the paper. This paper is structured as follows. Section 2 recalls the sequent calculus and presents the proof systems implemented in GenZ. Section 3 recalls the zipper data structure. In Section 4 we describe our implementation, and illustrate how users can define their own proof systems in it. Our experimental evaluation is presented in Section 5.

2. Sequent calculi for modal logics

We first present the logics and the proof systems implemented in GenZ, namely sequent calculus for classical propositional logic (CPL), intuitionistic propositional logic (IPL), calculi for (some) modal logics in the S5-cube (cf. Figure 1), and for Gödel-Löb logic (GL). GenZ supports cut-free proof systems. Consequently, we currently cover K, K4, K45, D, D4, D45, T and S4. For the moment, we do not support logics K5, D5, or any logic featuring axiom b, because the sequent calculi for these logics include a cut rule.

Given a set of countably infinite propositional variables, the propositional language $\mathcal{L}$ is generated as follows, where p is a propositional atom:

$$\varphi ::= p \mid \bot \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \to \varphi$$

Similarly, the modal language $\mathcal{L}^{\Box}$ is generated by:

$$\varphi ::= p \mid \bot \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \to \varphi \mid \Box\varphi$$

Other operators are definable as usual, e.g., $\neg\varphi := \varphi \to \bot$ and $\Diamond\varphi := \neg\Box\neg\varphi$. For convenience, we choose the language $\mathcal{L}$ to be suitable for both classical and propositional logic. When working with classical systems, one can restrict the number of propositional operators. We will introduce a two-sided sequent calculus, suitable for both classical and intuitionistic systems. As a consequence, $\Diamond$ is taken as definable, to reduce the number of rules. All these choices can be reverted: the grammar of GenZ can easily be changed to employ other operators as primitives.

Hilbert-style axiom systems of CPL and IPL can be found, e.g., in [26]. Axiom systems for modal logics extend an axiomatization of CPL with the axioms and rules from Figure 1; see also [27]. Finally, an axiomatisation of GL is obtained by adding to an axiomatisation of K the following axiom, called gl: $\Box(\Box\varphi \to \varphi) \to \Box\varphi$ (so $\mathsf{GL} = \mathsf{CPL} \cup \{\mathsf{k}, \mathsf{nec}, \mathsf{gl}\}$, refer to Figure 1).

A *sequent* is a pair $\Gamma \Rightarrow \Delta$, where Γ, Δ are *sets* of formulas of $\mathcal{L}$ or of $\mathcal{L}^{\Box}$. Using sets to represent sequents is not essential, but significantly improves the performance of GenZ, since it makes the detection of repeated sequents easy. As usual, a sequent can be interpreted as a formula of the language, namely: $\bigwedge\Gamma \to \bigvee\Delta$.

We first discuss the calculi for the logics $\mathsf{L} \in \{\mathsf{CPL}, \mathsf{IPL}, \mathsf{K}, \mathsf{K4}, \mathsf{K45}, \mathsf{D}, \mathsf{D4}, \mathsf{D45}, \mathsf{T}, \mathsf{S4}\}$, that is, all the logics implemented by GenZ, except GL. For each logic L, we write $\mathsf{Seq}_{\mathsf{L}}$ to denote its sequent

k	$\Box(\varphi \to \psi) \to (\Box\varphi \to \Box\psi)$
nec	If $\vdash \varphi$, then $\vdash \Box\varphi$
d	$\Box\varphi \to \Diamond\varphi$
t	$\Box\varphi \to \varphi$
b	$\varphi \to \Box\Diamond\varphi$
4	$\Box\varphi \to \Box\Box\varphi$
5	$\Diamond\varphi \to \Box\Diamond\varphi$

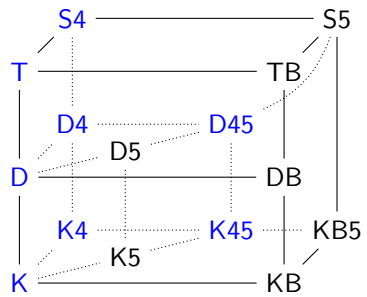


Figure 1: Modal logics in the S5 cube and their axioms. Logic K is axiomatised by $\mathsf{CPL} \cup \{\mathsf{k}, \mathsf{nec}\}$. The other systems are axiomatised by adding to K the axioms in $\{\mathsf{d}, \mathsf{t}, \mathsf{b}, 4, 5\}$ corresponding to the name and position of the logic in the modal cube. For instance, $\mathsf{S4} = \mathsf{K} \cup \{\mathsf{t}, 4\}$. GenZ implements sequent calculi for logics in blue.

$$\mathsf{ax}\ \frac{}{\Gamma, p \Rightarrow \Delta, p} \qquad \bot_{\mathsf{L}}\ \frac{}{\Gamma, \bot \Rightarrow \Delta} \qquad \wedge_{\mathsf{L}}\ \frac{\Gamma, \varphi, \psi \Rightarrow \Delta}{\Gamma, \varphi \wedge \psi \Rightarrow \Delta} \qquad \vee_{\mathsf{R}}\ \frac{\Gamma \Rightarrow \Delta, \varphi, \psi}{\Gamma \Rightarrow \Delta, \varphi \vee \psi}$$

$$\wedge_{\mathsf{R}}\ \frac{\Gamma \Rightarrow \Delta, \varphi \qquad \Gamma \Rightarrow \Delta, \psi}{\Gamma \Rightarrow \Delta, \varphi \wedge \psi} \qquad \vee_{\mathsf{L}}\ \frac{\Gamma, \varphi \Rightarrow \Delta \qquad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \varphi \vee \psi \Rightarrow \Delta}$$

$$\rightarrow_{\mathsf{L}}\ \frac{\Gamma \Rightarrow \Delta, \varphi \qquad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \varphi \rightarrow \psi \Rightarrow \Delta} \qquad \rightarrow_{\mathsf{R}}\ \frac{\Gamma, \varphi \Rightarrow \Delta, \psi}{\Gamma \Rightarrow \Delta, \varphi \rightarrow \psi}$$

$$\rightarrow^{i}_{\mathsf{L}}\ \frac{\Gamma, \varphi \rightarrow \psi \Rightarrow \Delta, \varphi \qquad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \varphi \rightarrow \psi \Rightarrow \Delta} \qquad \rightarrow^{i}_{\mathsf{R}}\ \frac{\Gamma, \varphi \Rightarrow \psi}{\Gamma \Rightarrow \Delta, \varphi \rightarrow \psi}\ \not\emptyset$$

$$\Box_{\mathsf{k}}\ \frac{\Gamma \Rightarrow \varphi}{\Box\Gamma, \Sigma \Rightarrow \Delta, \Box\varphi} \qquad \Box_{\mathsf{d}}\ \frac{\Gamma, \varphi \Rightarrow}{\Box\Gamma, \Box\varphi, \Sigma \Rightarrow \Delta} \qquad \Box_{\mathsf{t}}\ \frac{\varphi, \Box\varphi, \Gamma \Rightarrow \Delta}{\Box\varphi, \Gamma \Rightarrow \Delta} \qquad \Box_{4}\ \frac{\Box\Gamma, \Gamma \Rightarrow \varphi}{\Box\Gamma, \Sigma \Rightarrow \Delta, \Box\varphi}\ \not\emptyset$$

$$\Box_{\mathsf{k45}}\ \frac{\Box\Gamma_1, \Gamma_2 \Rightarrow \Box\Delta, \varphi}{\Box\Gamma_1, \Box\Gamma_2, \Sigma \Rightarrow \Box\Delta, \Box\varphi, \Pi}\ \not\emptyset \qquad \Box_{\mathsf{d45}}\ \frac{\Box\Gamma_1, \Gamma_2 \Rightarrow \Box\Delta}{\Box\Gamma_1, \Box\Gamma_2, \Sigma \Rightarrow \Box\Delta, \Pi}\ \not\emptyset$$

Figure 2: The rules of $\mathsf{Seq}_{\mathsf{CPL}}$ are displayed on the *top*. The rules of $\mathsf{Seq}_{\mathsf{IPL}}$ are the rules of $\mathsf{Seq}_{\mathsf{CPL}}$, with rules $\rightarrow_{\mathsf{L}}$ and $\rightarrow_{\mathsf{R}}$ replaced by $\rightarrow^{i}_{\mathsf{L}}$ and $\rightarrow^{i}_{\mathsf{R}}$ are shown in the **middle**. At the **bottom** we display the sequent calculus rules for modal logics, where $\mathsf{Seq}_{\mathsf{K}} = \mathsf{Seq}_{\mathsf{CPL}} \cup \{\Box_{\mathsf{k}}\}$; $\mathsf{Seq}_{\mathsf{D}} = \mathsf{Seq}_{\mathsf{K}} \cup \{\Box_{\mathsf{d}}\}$; $\mathsf{Seq}_{\mathsf{T}} = \mathsf{Seq}_{\mathsf{K}} \cup \{\Box_{\mathsf{t}}\}$; $\mathsf{Seq}_{\mathsf{K4}} = \mathsf{Seq}_{\mathsf{CPL}} \cup \{\Box_{4}\}$; $\mathsf{Seq}_{\mathsf{D4}} = \mathsf{Seq}_{\mathsf{K4}} \cup \{\Box_{\mathsf{d}}\}$; $\mathsf{Seq}_{\mathsf{S4}} = \mathsf{Seq}_{\mathsf{K4}} \cup \{\Box_{\mathsf{t}}\}$; $\mathsf{Seq}_{\mathsf{K45}} = \mathsf{Seq}_{\mathsf{CPL}} \cup \{\Box_{\mathsf{k45}}\}$; $\mathsf{Seq}_{\mathsf{D45}} = \mathsf{Seq}_{\mathsf{K45}} \cup \{\Box_{\mathsf{d45}}\}$. Symbol $\not\emptyset$ denotes *global loop check*.

calculus. The rules of $\mathsf{Seq}_{\mathsf{CPL}}$ and $\mathsf{Seq}_{\mathsf{IPL}}$ are shown on the top and the center in Figure 2. The sequent calculus $\mathsf{Seq}_{\mathsf{CPL}}$ is the calculus **G3cp** from [26], while $\mathsf{Seq}_{\mathsf{IPL}}$ is the multi-conclusion Maehara-style calculus for IPL, denoted **mG3ip** in [26] (there are many different versions of sequent calculi for CPL and IPL: we refer the reader to [26]). The rules of $\mathsf{Seq}_{\mathsf{L}}$ are obtained by extending $\mathsf{Seq}_{\mathsf{CPL}}$ with the modal rules from the bottom of Figure 2. Here, for a set Γ of formulas, we let $\Box\Gamma = \{\Box\varphi \mid \varphi \in \Gamma\}$. For an overview of modal sequent calculi, see [28].

For every logic in L, a *derivation* of a sequent $\mathcal{S}$ in $\mathsf{Seq}_{\mathsf{L}}$ is a finite tree whose root is labeled with $\mathcal{S}$, whose leaves are labeled with sequents derived by rules ax or $\bot_{\mathsf{L}}$, and whose intermediate nodes are labeled with sequents generated via the inference rules of $\mathsf{Seq}_{\mathsf{F}}$. We say that a sequent $\mathcal{S}$ is *derivable* in $\mathsf{Seq}_{\mathsf{L}}$ if there is a derivation of $\mathcal{S}$ in $\mathsf{Seq}_{\mathsf{L}}$. A formula φ is then *provable* in $\mathsf{Seq}_{\mathsf{L}}$ if there is a derivation of $\Rightarrow \varphi$ in $\mathsf{Seq}_{\mathsf{L}}$. A *proof search tree* is a (possibly infinite) tree constructed by applying the rules of the calculus bottom-up to a sequent. The proof search algorithms implemented by GenZ construct proof search trees for a given sequent, trying to find a derivation for it.

All the proof systems $\mathsf{Seq}_{\mathsf{L}}$ absorb the structural rules of *weakening*, displayed below, within the logical rules:

$$\mathsf{wk}_{\mathsf{L}}\ \frac{\Gamma \Rightarrow \Delta}{\varphi, \Gamma \Rightarrow \Delta} \qquad \mathsf{wk}_{\mathsf{R}}\ \frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, \varphi}$$

More precisely, these rules are *admissible* in every system $\mathsf{Seq}_{\mathsf{L}}$, where we say that a rule R is admissible in a proof system R if, whenever the premises of R are derivable in the proof system, its conclusion is also derivable. Similarly, the cut rule, displayed below, is admissible in every calculus $\mathsf{Seq}_{\mathsf{L}}$:

$$\mathsf{cut}\ \frac{\Gamma \Rightarrow \Delta, \varphi \qquad \varphi, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta}$$

While the cut rule is helpful in proving completeness, as it simulates *modus ponens*, it is desirable to work with a cut-free sequent calculus, especially if one wants to implement root-first proof search. Indeed, in any application of the cut rule, the cut formula φ needs to be guessed when going from the conclusion to the premises of the rule. This makes a generic treatment of proof systems with cut as a primitive rule challenging. By showing admissibility of the cut rule, one proves that the rule can be safely removed from the calculus, while still maintaining completeness.

All the rules of $\mathsf{Seq}_{\mathsf{CPL}}$ are *invertible*, meaning that the derivability of the conclusion of a rule implies the derivability of each premise of the rule. Building a single proof search tree is thus enough to check the derivability of a sequent in $\mathsf{Seq}_{\mathsf{CPL}}$. In contrast, the rules $\rightarrow^{i}_{\mathsf{L}}$ and $\rightarrow^{i}_{\mathsf{R}}$ of $\mathsf{Seq}_{\mathsf{IPL}}$ are not invertible,

$$\to^i_{\mathsf{R}} \dfrac{\times \dfrac{}{p, q \Rightarrow s}}{p \Rightarrow q \to s, q \to p} \qquad \to^i_{\mathsf{R}} \dfrac{\mathsf{ax} \dfrac{}{p, q \Rightarrow p}}{p \Rightarrow q \to s, q \to p} \qquad \to_{\mathsf{R}} \dfrac{\to_{\mathsf{R}} \dfrac{\mathsf{ax} \dfrac{}{p, q \Rightarrow s, p}}{p, q \Rightarrow s, q \to p}}{p \Rightarrow q \to s, q \to p}$$

Figure 3: An example of why backtracking is needed. We try to prove the sequent $p \Rightarrow q \to s, q \to p$ in $\mathsf{Seq}_{\mathsf{IPL}}$, by constructing a proof search tree for it. First, we apply the non-invertible rule $\to^i_{\mathsf{R}}$ (**left**), and we are stuck. We thus need to backtrack and choose a different formula to which to apply rule $\to^i_{\mathsf{R}}$. The second choice gives us a derivation of the sequent (**middle**). By contrast, constructing one proof search tree is enough to prove the same sequent in $\mathsf{Seq}_{\mathsf{CPL}}$: even if we apply rule $\to^i_{\mathsf{R}}$ to the 'wrong' formula first, we still have the other one available, because rule $\to^i_{\mathsf{R}}$ is invertible.

and backtracking is needed to ensure that all possible proof search trees are explored (when applying rules $\to^i_{\mathsf{L}}$ and $\to^i_{\mathsf{R}}$ bottom-up, one might make a 'wrong' choice, refer to Figure 3). From the modal rules, $\Box_{\mathsf{t}}$ is the only invertible rule, so backtracking is needed when implementing proof search in all the modal proof systems.

GenZ implements terminating root-first proof search in $\mathsf{Seq}_{\mathsf{L}}$. Root-first proof search for $\mathsf{Seq}_{\mathsf{CPL}}$ terminates without the need for any additional loop check: the application of every rule decreases (bottom-up) the number of logical operators occurring in a sequent.

For the modal calculi, proof search in $\mathsf{Seq}_{\mathsf{K}}$ and $\mathsf{Seq}_{\mathsf{D}}$ terminates, as the bottom-up application of every rule decreases the total number of connectives in a sequent. To ensure termination in the other modal proof systems, GenZ employs two *loop checks*, each dealing with a specific kind of non-terminating behaviour.

First, rule $\Box_{\mathsf{t}}$ could be applied indefinitely to a sequent, because of the repetition of formula $\Box\varphi$ in the premise of the rule. To prevent this, we adopt a *cumulative* version of the propositional rules in $\mathsf{Seq}_{\mathsf{T}}$ and $\mathsf{Seq}_{\mathsf{S4}}$. That is, we repeat the principal formula in the premise of every propositional rule. For instance, the following are cumulative versions of $\wedge_{\mathsf{L}}$ and $\to_{\mathsf{L}}$:

$$\wedge_{\mathsf{L}} \dfrac{\Gamma, \varphi \wedge \psi, \varphi, \psi \Rightarrow \Delta}{\Gamma, \varphi \wedge \psi \Rightarrow \Delta} \qquad \to_{\mathsf{L}} \dfrac{\Gamma, \varphi \to \psi \Rightarrow \Delta, \varphi \qquad \Gamma, \varphi \to \psi, \psi \Rightarrow \Delta}{\Gamma, \varphi \to \psi, \varphi \to \psi \Rightarrow \Delta}$$

Then, we adopt a *local loop check*: a rule R cannot be applied to a sequent $\mathcal{S}$ if the formulas introduced in a premise of R already occur in $\mathcal{S}$. For instance, $\Box_{\mathsf{t}}$ can be applied to the sequent $\Box\varphi, \Gamma \Rightarrow \Delta$ only if $\varphi \notin \Gamma$. This loop check is performed before the application of every rule in $\mathsf{Seq}_{\mathsf{T}}$ and $\mathsf{Seq}_{\mathsf{S4}}$.

Second, rules $\Box_4$, $\Box_{\mathsf{k45}}$ and $\Box_{\mathsf{d45}}$ might repeat $\Box$-formulas in their premises. While there cannot be infinitely many *consecutive* applications of these rules, their interaction with other rules might generate infinite branches. To ensure termination in the calculi featuring these rules, GenZ employs a *global loop check* ($\not\emptyset$ in Figure 1), blocking proof search *before* applying one of the rule $\mathsf{R} \in \{\Box_4, \Box_{\mathsf{k45}}, \Box_{\mathsf{d45}}\}$ in case the sequent $\Gamma \Rightarrow \Delta$, which would be the premise of R, is a 'repetition' of a previous sequent, i.e., if there is a sequent $\Gamma' \Rightarrow \Delta'$ occurring lower and on the same branch of $\Gamma \Rightarrow \Delta$ such that $\Gamma \subseteq \Gamma'$ and $\Delta \subseteq \Delta'$. Moreover, GenZ applies rules in $\Box_4$, $\Box_{\mathsf{k45}}$ and $\Box_{\mathsf{d45}}$, which are not invertible, only after all the invertible rules have been non-redundantly applied. Observe that the global loop check makes the local loop check redundant. However, we choose to keep both, as having the local loop check still results in shorter proof search trees.

To ensure termination in $\mathsf{Seq}_{\mathsf{S4}}$, GenZ applies both the local and the global loop check. Something similar happens in the case of $\mathsf{Seq}_{\mathsf{IPL}}$. In this calculus, rule $\to^i_{\mathsf{L}}$ repeats formula $\varphi \to \psi$ in the leftmost premise of the rule (intuitively, we might need to apply this rule more than once in a branch of a derivation). Thus, this rule might be applied indefinitely to a sequent and, when combined with the non-invertible rule $\to^i_{\mathsf{R}}$, might lead to infinite branches. As in the modal case, GenZ adopts a *cumulative* version of $\mathsf{Seq}_{\mathsf{IPL}}$ (the cumulative version of rule $\to^i_{\mathsf{L}}$ is the same as the cumulative version of $\to_{\mathsf{L}}$, displayed above). Then, the local loop check dictates that a rule can be applied bottom-up to a sequent only if some formulas which are not present in the conclusion are introduced in one of the premises of the rule. The global loop check is instead applied only before rule $\to^i_{\mathsf{R}}$, to check that no repeated

sequents are present in the branch. The non-invertible rule $\to^i_{\mathsf{R}}$ is applied only after all the other rules have been non-redundantly applied.

Finally, let us briefly discuss the proof system $\mathsf{Seq}_{\mathsf{GL}}$ for Gödel-Löb logic GL, which showcases the ability of GenZ to handle *cyclic* sequent calculi.[1] As shown in [31], a cyclic proof system for GL (which we call $\mathsf{Seq}_{\mathsf{GL}}$) can be defined by allowing cycles in $\mathsf{Seq}_{\mathsf{K4}}$, the sequent calculus proof system for K4 (refer to Figure 1). The rules of $\mathsf{Seq}_{\mathsf{GL}}$ are those of $\mathsf{Seq}_{\mathsf{K4}}$, but proofs in $\mathsf{Seq}_{\mathsf{GL}}$ can be infinite objects. Namely, a *cyclic proof* in $\mathsf{Seq}_{\mathsf{GL}}$ is a possibly infinite but *regular* derivation tree $\mathcal{D}$ constructed with the rules of $\mathsf{Seq}_{\mathsf{K4}}$ (without $\circlearrowleft$). The regularity condition means that $\mathcal{D}$ contains only finitely many isomorphic subtrees. Intuitively, along every branch of $\mathcal{D}$, sequents will eventually repeat. The global loop check (modified to check for *equality* of sequents) can be used to detect regularity among infinite branches of a cyclic proof. In $\mathsf{Seq}_{\mathsf{GL}}$, a branch is closed if we detect a repetition in a valid branch. Such branches can be unfolded into an infinite branch by iterating the subtree identified by $\circlearrowleft$. For example, the following is a cyclic proof of axiom gl in $\mathsf{Seq}_{\mathsf{GL}}$ generated by GenZ.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Box a \to a, \Box(\Box a \to a) \Rightarrow a, \Box a}\,\circlearrowleft}{\Box a \to a, \Box(\Box a \to a) \Rightarrow a}\,\to^i_{\mathsf{L}}}{\Box(\Box a \to a) \Rightarrow a, \Box a}\,\Box_4}{\Box a \to a, \Box(\Box a \to a) \Rightarrow a}\,\to_{\mathsf{L}} \qquad \dfrac{}{a, \Box(\Box a \to a) \Rightarrow a}\,\mathsf{ax}}{\Box(\Box a \to a) \Rightarrow \Box a}\,\Box_4}{\Rightarrow \Box(\Box a \to a) \to \Box a}\,\to_{\mathsf{R}}$$

The formula $\Box(\Box a \to a) \to \Box a$ is not valid in K4, and indeed, proof search in $\mathsf{Seq}_{\mathsf{K5}}$ for this formula stops at the rule marked with $\circlearrowleft$. The above proof reflects the current implementation of GenZ where we check for repetitions only at the same place at which we would check for the global loop check. A slight optimisation would be to eagerly check for repetitions in $\mathsf{Seq}_{\mathsf{GL}}$.

The local and global proof search are well-known in the proof-theoretic literature (refer, e.g., to [26, 32]), and will be enough to ensure termination of several other *cut-free* sequent calculi the user might specify. Sequent calculi for the other modal logics in the cube, such as S5 [33], include an *analytic* cut rule:

$$\mathsf{a\text{-}cut}\ \dfrac{\Gamma \Rightarrow \Delta, \varphi \qquad \varphi, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta}\ (\varphi \textit{ is subformula of some formula in } \Gamma \cup \Delta)$$

The rule can be implemented in GenZ and, in principle, termination of proof search still holds. But the number of backtracking points increases dramatically, and additional heuristics are needed to implement effective search with a-cut.

3. The zipper

Central to our implementation is the use of the zipper to optimize memory usage. The zipper is a family of data structures which allow for efficient navigation of hierarchical structures, such as lists and trees, as well as for effective modification of those structures, exploiting pointers, or *focused positions*, within the structure. A zipper provides a means to navigate a data structure and edit its elements at a given location. The data structure is split into two parts: the *focus*, where modifications can be made, and the *context*, or *path*, which remains unchanged. The zipper was introduced by Gérard Huet in [22], who wrote: "The tree is turned inside-out like a returned glove, [...] Going up and down in the structure is analogous to closing and opening a zipper in a piece of clothing, whence the name."

The zipper can be defined for many data structures. We use them to encode proof search trees, but we first give a simpler example, the *zip list*. GenZ is implemented in Haskell, which is why we illustrate

[1] Several proof systems for GL have been defined, including a regular sequent calculus, introduced in [29], later corrected in [30]. This calculus could also be easily implemented in GenZ.

the following in this language. In Haskell, `[1,1,2,3,8,13]` is a list of integers and `"HellWorld"` is a list of characters. Irrespective of the length of the list, operations involving the first elements of a list are fast, while manipulating the innermost elements can be expensive. For example, to insert `5` between `3` and `8` in our list of integers, we first have to traverse the list until we reach the right place. Similarly, to insert the missing letter `o` at the right place into our list of characters, we need to walk past `"Hell"` first. A `ZipList` now represents a list by splitting it into two parts, and storing a *focus* that belongs in the middle:

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

Our example lists are "`Zip [2,1,1] 3 [8,13]`" and "`Zip "leH" 'l' "World"`". Note that we store the elements before the focus in reverse order. We can then define functions which create a singleton `ZipList`, output the value at the focus, insert or delete elements, and move the focus to the left or to the right:

```
singleton x = Zip [] x []
get       (Zip    _   p    _  ) = p
insert y  (Zip    xs  x    ys ) = Zip xs x (y:ys)
delete    (Zip    xs  _ (y:ys)) = Zip xs y    ys
moveLeft  (Zip (x:xs) p    ys ) = Zip xs x (p:ys)
moveRight (Zip    xs  p (y:ys)) = Zip (p:xs) y ys
```

A key feature of the zipper is that all these operations are constant-time, i.e. do not depend on the number of elements already stored.

Trees are hierarchical and non-linear data structures, more complex than lists. A standard definition of tree data structures in Haskell is:

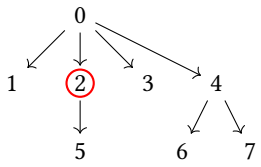
```
data Tree a = Node a [Tree a]
```

Thus, a tree is a node containing a value of type `a` and a list of children, which themselves are trees. Analogously to lists, operations near the root of a tree are fast, but operations deeper down near the leaves can be expensive. To remedy this, the *zip tree* works similarly to the *zip list*, by focusing on a node with its subtree and separately storing the parts "above" and "next to it".

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

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

A value of type `Path` is either `Top` to say that we are at the root, or it is a `Step` upwards, leading to the parent `a` value, the `Path a` above it, and left and right siblings `[Tree a]`. The following example shows a tree with the focus at node 2.




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

4. Implementation

GenZ is implemented in Haskell and provides two ways of representing proof search trees: one using standard trees and one using the zipper data structure. The purpose of the implementation using trees is to experimentally evaluate the impact of the zipper data structure on the performance of our system. The prover is *modular*: proof search is not implemented for any specific logic, but takes a proof system for a specific logic as input.

The prover can be used via a web interface or as a command-line tool. All source code and a readme with detailed information for end-users can be found in our repository. The exact version and the benchmark scripts we describe here have also been archived at Zenodo [34]. The command-line tool provides the full potential of our tool, as users can easily add their own proof systems as described

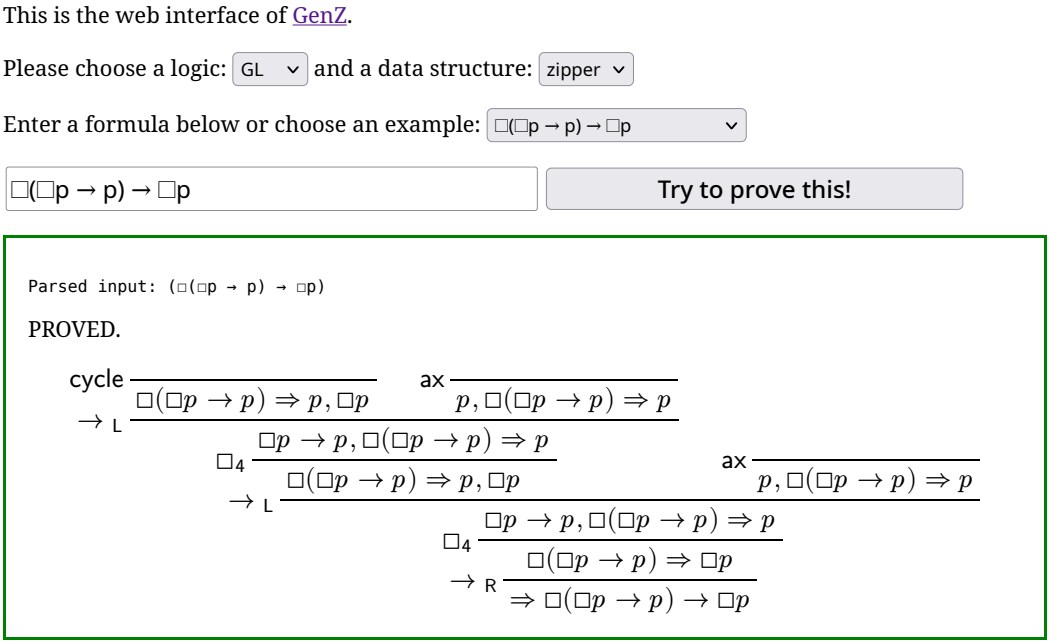


Figure 4: Screenshot of the web interface of GenZ.

below. The web interface can be used to try out the already implemented proof systems. We show a screenshot of the web interface in Figure 4, where a proof of $((p \vee (p \to \bot)) \to \bot) \to \bot$ in $\mathsf{Seq}_{\mathsf{IPL}}$ has been generated. Proof systems are specified using Haskell code. We explain in the following how this is done, and then describe our main algorithm for generating proofs for a given proof system.

4.1. Data structures for formalizing proof systems

As discussed in Section 2, a *sequent* has the form $\Gamma \Rightarrow \Delta$, where Γ and Δ are sets of formulas. Sequents are encoded using the Haskell module `Data.Set`. Using sets to represent sequents is not essential—one could also use lists for example, but significantly improves the performance, since repeated sequents can be fast detected in this data structure.

We then represent a *rule* as a function that takes three arguments: the history (i.e. current branch), the current sequent and a principal formula in the sequent.

```
type Sequent f = Set (Either f f)
type RuleName = String
type Rule f = History f -> Sequent f -> Either f f -> [(RuleName, [Sequent f])]
```

The return type `[(RuleName, [Sequent f])]` represents all possible applications of a rule to a principal formula. Each result is a pair of a rule name, such as `"□k"`, and a list of resulting sequents, allowing for branching (e.g. in the case of propositional rules in Figure 2). Simple replacement rules that do not depend on the history can also be defined using the `replaceRule` helper function.

A `Logic` is a collection of rules. As argued in Section 2, non-invertible rules introduce a form of non-determinism. In GenZ, invertible rules are called *safe*, and non-invertible ones *unsafe*. The rules of a `Logic` are correspondingly divided into `safeRules` (invertible rules, which may be greedily applied in any order) and `unsafeRules` (non-invertible rules, that may need backtracking).

```
data Logic f =
  Log { name :: String, safeRules :: [Rule f], unsafeRules :: [Rule f] }
```

GenZ is fully generic in the sense that users only have to provide an instantiation of `Logic` for the prover to work.

4.2. Proof generation in GenZ

We illustrate how proof systems are formalized on two examples. Our implementation includes formalizations of the 11 proof systems discussed in Section 2. We first illustrate the proof system for classical propositional logic (CPL). Since all the rules in $\mathsf{Seq}_{\mathsf{CPL}}$ are invertible, $\mathsf{Seq}_{\mathsf{CPL}}$ has no unsafe rules.

```
module Logic.Propositional.CPL (classical) where

import qualified Data.Set as Set
import General
import FormP

classical :: Logic FormP
classical = Log { name = "CPL"
                , safeRules   = [leftBot, isAxiom, replaceRule safeCPL]
                , unsafeRules = [] }

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

The rules `leftBot` and `isAxiom` are provided as standard rules and do not need to be specified by the user. All other rules of $\mathsf{Seq}_{\mathsf{CPL}}$ are implemented as a single rule `safeCPL`, which is a `replaceRule`, that is, it does not depend on the history. The different cases correspond to the different rules on top of Figure 2. For example, the $\wedge_{\mathsf{L}}$-rule is implemented in the first case and picks a conjunction "`(ConP f g)`" from the left sequent (denoted using the keyword `Left`), which in the premise of the rule is replaced by the two formulas `g` and `f` on the left, constructed as a set from the list `[Left g, Left f]`.

To illustrate the use of unsafe rules and treatment of special termination conditions, we provide as the second example the formalization of the calculus $\mathsf{Seq}_{\mathsf{GL}}$ for GL, as an extension of the one for K. Here we also have a non-invertible rule that is contained in `unsafeRules`.

```
module Logic.Modal.GL (gl) where

import qualified Data.Set as Set
import Logic.Modal.K
import General
import FormM

gl :: Logic FormM
gl = Log { name = "GL"
         , safeRules   = [leftBot, isAxiom, replaceRule safeML, isCycle]
         , unsafeRules = [box4rule] }

isCycle :: Rule FormM
isCycle h fs _ = [("cycle", []) | fs `elem` h]

-- | The 4 box rule: without global loop check
box4rule :: Rule FormM
box4rule _ fs (Right (Box f)) = concatMap func ss where
  func :: Sequent FormM -> [(RuleName,[Sequent FormM])]
  func seqs = [("□4", [seqs])]
  ss = Set.map (\s -> Set.unions [Set.singleton (Right f), s, Set.map fromBox s]) ss'
  ss' = Set.powerSet $ Set.filter isLeftBox fs
box4rule _ _ _ = []
```

Here, `isCycle` checks whether a sequent is repeated on the current branch, and then closes this branch. The $\Box_4$ is implemented in `box4rule`, which, produces all possible ways of applying the rule. For this,

Algorithm 1: `extendZ`

Input: logic L and `ZipProof` with focus `fs` and path `p`
Output: a list of `ZipProofs`

```
if any safeRule of L is applicable to some φ in fs then
    apply the rule to create children
    move focus down to the first (left-most) child
    call extendZ again at the new focus
results := [] // In Haskell: build lazy list 'List.concatMap applyRule rs'
foreach unsafeRule r of L do
    foreach φ in fs to which r is applicable do
        // for simplicity, assume r is non-branching
        foreach sequent gs resulting from r applied to φ in fs do
            next := gs as child of fs
            if any element of extendZ next is closed then
                add the first such element to results
            else
                add fs to results // backtracking
return results
```

the variable `ss'` collects all the possible subsets of the left side of the sequent that use the box operator, for which the function `isLeftBox` is used. The $\Box_4$ is an unsafe rule, which for GL and in the presence of the cycle detection does not require a global loop check. That is, unlike in $\mathsf{Seq}_{\mathsf{K4}}$ where a repeated sequent blocks proof search, in $\mathsf{Seq}_{\mathsf{GL}}$ a repetition closes the branch successfully (cf. Figure 2).

4.3. Generating proofs

To encode a *proof*, we define two options: standard trees and the zipper. For standard trees, each `Node` contains a `Maybe` value indicating if the node has been extended, and a `Bool` marker for already closed proofs. The `RuleName` indicates which rule was applied to go from this node to the children. The type `ProofWithH` (for proof with history) is used for loop checks.

```
data Proof f = Node (Sequent f) (Maybe (RuleName, [Proof f])) Bool
newtype ProofWithH f = HP (History f, Proof f)
```

Getting to our most important type definition, a `ZipProof` represents a proof tree with a focus. Together with `ZipPath`, it can replace `ProofWithH`. Note the similarity to `ZipTree` and `Path` from page 3.

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

We finally turn to the *proof search*, which we implement for both `ProofWithH` and `ZipProof`. When running GenZ, the user can choose the data structure. Here we focus on the zipper version, which is also the default. To check whether a formula φ is provable, we start from the sequent $\Rightarrow \varphi$, i.e., the initial `ZipProof` value is `ZP (Node (Set.singleton (Right f)) Nothing False) Top`. The main work is done by the function `extendZ`, which tries to extend the current proof to a complete one. The function greedily applies safe rules and returns a list of `ZipProof` values to allow backtracking for unsafe rules. We provide a pseudocode version of the function `extendZ` as Algorithm 1. For full details, we refer to the Haskell code, where the **foreach** loops are in fact implemented by `map` with laziness.

Name	Size	Formula	
`pei`	-	$((p \to q) \to p) \to p$	
`conPeiR`	n	$\texttt{pei} \wedge (\texttt{pei} \wedge \cdots \wedge (\texttt{pei} \wedge \texttt{pei}))$	($2n+1$ occurrences of `pei`)
`conPeiL`	n	$((\texttt{pei} \wedge \texttt{pei}) \wedge \cdots \wedge \texttt{pei}) \wedge \texttt{pei}$	($2n+1$ occurrences of `pei`)
`multiVerK`	n	$\Box(p_n \to \ldots \to (p_2 \to p_1)) \to (\Box p_n \to \ldots \to (\Box p_2 \to \Box p_1))$	
`lobBoxes`	n	$\Box(\Box p \to p) \to \Box \cdots \Box p$	(n $\Box$'s in consequent)

Figure 5: Custom-made benchmark formulas, where $n \in \mathbb{N}$ and $p, q, p_1, \ldots, p_n$ are propositional atoms.

5. Experimental evaluation

We evaluated our system to 1) understand the limits and feasibility of our approach and 2) to measure the impact of the zipper structure. Our aim is not to compete with highly-optimized state-of-the-art reasoners, and we do not think that GenZ is a suitable choice to work on larger-scale reasoning problems. Instead, we think that the main value of GenZ is to help in education and research with a tool that allows one to quickly try and compare different (existing and new) proof systems. Consequently, we focus our evaluation on smaller formulas whose tree size does not exceed 200, and we do not do a comparative study with highly-optimized state-of-the-art systems, whose performance is out of range for us. To be helpful for our envisioned use cases, proofs should be generated ideally in a few seconds or less, and the resulting proofs should not become too large in case we want to look at them in practice. In addition to presenting a new generic prover, a contribution of our paper is also the idea of using a zipper data structure. To measure the effect, we compared our two implementations—one using zippers and one using standard trees—on systematically generated formulas.

We evaluated the performances of GenZ on the 11 implemented sequent calculi, using custom-made formulas, formulas from the Logics Workbench (LWB) [24], and from the ILTP library [25]. All experiments were run on the Dutch national supercomputer *Snellius*, operated by SURF. We used a single thin compute node on the *Genoa* partition, equipped with 2×96-core AMD EPYC 9654 CPUs (192 cores in total, 1.5–3.7 GHz) and 384 GiB RAM, running Linux. GenZ itself is single-threaded. Therefore, our results reflect single-core performance on a modern CPU with comparable single-core speed.

5.1. Impact of the zipper

To measure the impact of the zipper data structure on memory consumption, we first ran the tree and zipper implementations on systematically generated formulas. We specified formulas of increasing size, including both provable and non-provable formulas. The set of formulas is shown in Figure 5 Here, `pei` stands for Peirce's law, which is valid in CPL and not valid in IPL. Similarly, both `ConPeiR` and `ConPeiL` are valid in CPL but not in IPL. Formula `multiVerK` is valid in K, and formula `lobBox` is valid only in GL. For example, formula `conPeiR 2` is $\texttt{pei} \wedge (\texttt{pei} \wedge (\texttt{pei} \wedge \texttt{pei}))$ whereas `lobBoxes 3` is $\Box(\Box p \to p) \to \Box\Box\Box p$.

We used the Haskell library *Weigh* [35] to track memory usage and the number of garbage collections (GCs). We show the results for these four formulas in Table 1. For non-provable formulas, we used substantially less memory with the zipper data structure than with the tree, as well as fewer or no garbage collections. This confirmed our hypothesis that using the zipper allows for more space-efficient reasoning. However, the tree-based and zipper-based implementations employed similar memory usage on provable modal formulas.

Part of the reason why our zipper prover outperforms the tree prover is also that the former implements an early-fail strategy, which one can, in theory, also implement with a tree data structure. The zipper naturally employs a branch-by-branch search strategy: it extends one branch at a time, and if that branch fails, it backtracks immediately without exploring siblings. In contrast, the tree prover explores both children and combines results using "pickOneOfEach", which is the idiomatic approach for that data structure. While one could implement a similar early-fail strategy for trees, doing so would

Logic	Formula		Size	Result	Allocated bytes	GCs
IPL	`conPeiL`	tree	100	False	124,670,912	29
IPL	`conPeiL`	zipper	100	False	53,048,840	12
IPL	`conPeiR`	tree	100	False	124,708,432	29
IPL	`conPeiR`	zipper	100	False	22,696	0
K	`multiVerK`	tree	20	True	85,699,081,760	20,494
K	`multiVerK`	zipper	20	True	85,699,064,824	20,494
K4	`lobBoxes`	tree	10	False	13,588,888	3
K4	`lobBoxes`	zipper	10	False	240,576	0
GL	`lobBoxes`	tree	100	True	24,179,016	5
GL	`lobBoxes`	zipper	100	True	25,517,216	6

Table 1
Memory usage.

require additional bookkeeping, whereas the zipper provides this functionality as standard through its focus structure. This is precisely the point of our comparison: the zipper allows efficient backtracking naturally. We are not comparing two identical algorithms on different data structures, but rather how each data structure shapes the proof search strategy it naturally admits.

5.2. Performance on the Logics Workbench

Next, we evaluated the performance of GenZ on formulas by existing benchmarks, to get a general understanding of its limitations. We started with formulas from the LWB suite in the version provided in [36]. In this version, each formula is the negation of the original LWB formula. [2] To convert these satisfiability tasks into provability tasks, we negated each formula again, thus retrieving the original LWB formulas.

Our aim is not to compete with highly optimized state-of-the-art provers, but rather to provide a system that simplifies exploring different sequent calculi for educational or research purposes, and to generate user-understandable proofs. Consequently, we favoured a clean and transparent over a highly optimized implementation. The downside is that our system can, in most cases, not deal with formulas of size larger than 200, as we found out in our first experiments on those benchmarks. Here, *size* corresponds to *tree size*, which is the number of atoms and symbols needed to write down the formula. To obtain a clearer understanding, we restricted the benchmark to formulas of a size below 200. This resulted in a set of 49 formulas, around 13% of the formulas in the LWB benchmark. The formulas had an average size of 92, with a median of 83. The size distribution of these formulas is shown in Figure 6.

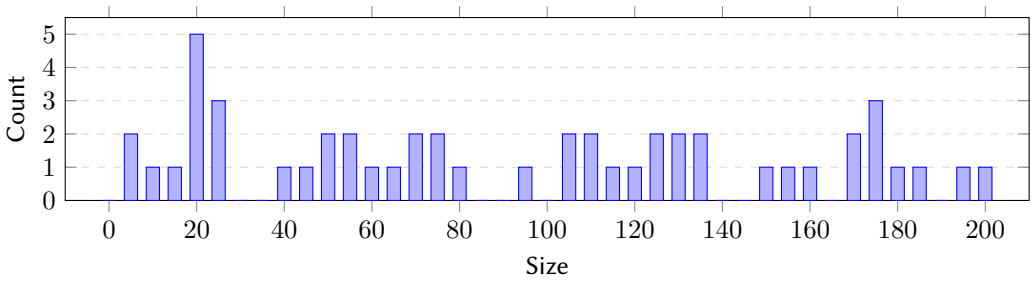


Figure 6: Size distribution of LWB formulas (restricted to size < 200).

[2] The original website and source files of the LWB benchmark [24] are no longer available. We took the benchmark formulas from [36], which provided provable and non-provable formulas for K and S4 only. These formulas constitute our LWB benchmark suite.

	All	K	K4	D	D4	T	S4	GL	K45	D45
zipper	86.85	97.96	87.76	97.96	85.71	87.76	71.43	91.84	93.88	67.35
tree	80.50	100.00	87.76	77.55	63.27	87.76	65.31	93.88	95.92	53.06

Table 2
Success rates of proving formulas in the benchmark for the different sequent calculi, using a timeout of 300 seconds.

	All				Provable				Non-provable			
	median	mean	p_{75}	p_{90}	median	mean	p_{75}	p_{90}	median	mean	p_{75}	p_{90}
zipper	0.01	40.64	0.05	300	0.01	3.96	0.05	2.99	0.01	0.08	0.01	0.04
tree	0.01	60.44	1.85	300	0.01	5.63	0.03	2.23	0.01	0.68	0.01	0.09

Table 3
Runtime statistics for all logics. All values in seconds; p_{75} and p_{90} stand for 75th and 90th percentile, respectively.

We attempted to prove each formula with each of the nine modal calculi, resulting in 441 experimental runs, each with a timeout of 300 seconds. Table 2 shows the corresponding success rates. For most calculi, our success rates were above 85%, with the exceptions of $\mathsf{Seq}_{\mathsf{S4}}$ and $\mathsf{Seq}_{\mathsf{D45}}$. These systems include a global loop check which, in combination with the many backtrack points generated by rules $\Box_4$, $\Box_{\mathsf{k45}}$ and $\Box_{\mathsf{d45}}$, increases the time needed to (dis-)prove formulas. On $\mathsf{Seq}_{\mathsf{D}}$, $\mathsf{Seq}_{\mathsf{D4}}$ and $\mathsf{Seq}_{\mathsf{D45}}$, we observed a significant benefit from using the zipper data structure, which we conjecture is due to the zipper's more efficient handling of backtrack points in proof search.

Table 3 shows statistics on the runtimes of GenZ over all the LWB formulas in our benchmark, for all nine modal calculi. Table 4 reports runtime statistics per logic: each entry gives median/mean/p_{75}/p_{90} (in seconds) over all runs (All), provable runs (Thm, `res=True`), and non-provable runs (NonThm, `res=False`). Proving or disproving a formula took an average of 40.64 seconds using the zipper, but this number was heavily influenced by the timeouts and a few harder formulas, as 75% of formulas could be (dis-)proved in less than 0.05 seconds. Focusing only on formulas we could prove or disprove, 90% of them could be (dis-)proved in less than 2.99/0.04 seconds, indicating that many formulas can practically be (dis-)proved in real time.

Table 5 summarises the distribution of runtimes for the zipper- and tree-based implementations over six time bins, again split into All/Thm/NonThm. Even in logics where the success rates (for non-timeout runs) are similar, the zipper-based prover tends to concentrate more runs in the smallest time bins, in particular when showing non-provability.

In our tests, we also measured the size of the generated proofs, where the *size* of a proof is the number of inference steps. Considering all LWB formulas evaluated, using all nine modal calculi (as in Table 3), the size of completed proofs ranges from 6 to 3,256,329. For the zipper-based implementation, the median and mean proof sizes were 73 and 189,773, respectively. Interestingly, the tree-based implementation led to different proofs, where the median and mean proof sizes were 61 and 143,222. Note that the significant difference between the median and the mean is caused by outliers resulting in very large proofs, while the median indicates that most proofs are still of a size that could be visualized to end-users.

We further observed that disproving a formula is usually faster than finding a proof, in particular for the zipper-based prover. This can be explained by the zipper's search strategy: it extends one branch at a time and can stop early when a branch fails, while the tree-based prover explores both children in parallel and combines them via `pickOneOfEach`. As argued already above, similar strategies could be implemented directly for trees, but the zipper representation makes such backtracking behaviour more natural to express and implement.

Logic		All				Thm				NonThm			
K	zipper	0.01/	6.58/	0.01/	0.02	0.01/	2.15/	0.01/	2.16	0.01/	0.02/	0.01/	0.01
	tree	0.01/	1.15/	0.01/	0.02	0.01/	2.52/	0.01/	2.52	0.01/	0.8/	0.01/	0.02
K4	zipper	0.01/	37.32/	0.01/	300	0.01/	3.15/	0.03/	7.35	0.01/	0.01/	0.01/	0.01
	tree	0.01/	37.44/	0.01/	300	0.01/	3.62/	0.03/	8.11	0.01/	0.06/	0.01/	0.01
D	zipper	0.01/	6.64/	0.01/	0.03	0.01/	1.65/	0.01/	0.02	0.01/	0.12/	0.01/	0.02
	tree	0.01/	67.98/	5.36/	300	0.01/	1.95/	0.01/	0.01	0.01/	0.23/	0.01/	0.04
D4	zipper	0.01/	43.7/	0.02/	300	0.01/	3.12/	0.01/	10.84	0.01/	0.12/	0.01/	0.02
	tree	0.01/	114.53/	300/	300	0.01/	21.18/	0.01/	21.19	0.01/	0.01/	0.01/	0.02
T	zipper	0.02/	36.92/	0.20/	300	0.01/	0.39/	0.08/	1.34	0.02/	0.07/	0.05/	0.19
	tree	0.06/	38.49/	1.97/	300	0.01/	0.66/	0.14/	1.31	0.07/	3.07/	0.65/	5.31
S4	zipper	0.07/	88.48/	300/	300	0.02/	6.17/	0.07/	4.56	0.01/	0.42/	0.12/	0.38
	tree	0.4/	106.34/	300/	300	0.02/	5.37/	0.07/	1.27	0.01/	0.27/	0.52/	0.86
GL	zipper	0.01/	28.84/	0.01/	56.90	0.02/	17.75/	2.03/	23.02	0.01/	0.01/	0.01/	0.01
	tree	0.01/	23.64/	0.01/	34.98	0.03/	19.7/	2.28/	49.96	0.01/	0.07/	0.01/	0.01
K45	zipper	0.01/	18.82/	0.01/	0.16	0.01/	1.46/	0.06/	0.28	0.01/	0.01/	0.01/	0.01
	tree	0.01/	13.45/	0.01/	0.71	0.01/	1.73/	0.03/	0.49	0.01/	1.04/	0.01/	0.01
D45	zipper	0.02/	98.49/	300/	300	0.01/	1.38/	0.04/	0.28	0.01/	0.23/	0.01/	0.12
	tree	0.62/	140.98/	300/	300	0.01/	0.09/	0.02/	0.30	0.01/	0.59/	0.01/	0.51

Table 4
Runtime statistics per logic on the LWB benchmarks.

	zipper			tree		
	All	Thm	NonThm	All	Thm	NonThm
[0.01,0.1)	342	101	241	306	98	208
[0.1,1)	18	8	10	20	10	10
[1,10)	12	7	5	16	7	9
[10,100)	10	10	0	11	7	4
[100,300)	1	1	0	2	2	0
Timeout	58	-	-	86	-	-

Table 5
Number of LWB benchmark runs per runtime bin (in seconds), split by prover and provability.

We note that our performance is still far from what is reached by the state-of-the-art modal logic provers. To give an example, while our study only considered the 13% smallest formulas of the LWB benchmark, in a study from [37], the modal prover K_SP could solve 63% of the satisfiable and 86% of the unsatisfiable formulas in the modal logics it supports within 100 seconds.

5.3. ILTP library

Finally, we evaluated $\mathsf{Seq}_{\mathsf{CPL}}$ and $\mathsf{Seq}_{\mathsf{IPL}}$ on the ILTP library [25]. We only considered the propositional formulas in the ILTP library and, as in the LWB experiments, we only considered formulas of size < 200. We used all propositional formulas of size < 200 for CPL, and left out 10 complex formulas for IPL, since the validity of these formulas in IPL is unknown [25].

For $\mathsf{Seq}_{\mathsf{CPL}}$, we tested 165 formulas. The average formula size was 68, and the median was 59. The zipper-based and tree-based provers behaved similarly. They both achieved a success rate of almost 100%. The zipper-based prover had a mean runtime of 3.89 seconds and a median of 0.01 seconds, while

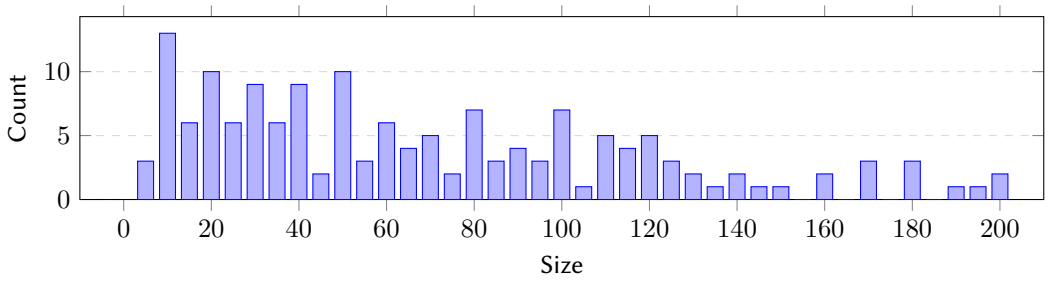


Figure 7: Size distribution of ILTP formulas used for $\mathsf{Seq}_{\mathsf{IPL}}$ (restricted to size < 200).

	zipper			tree		
	All	Thm	NonThm	All	Thm	NonThm
[0.01,0.1)	210	178	32	207	178	29
[0.1,1)	35	24	11	27	24	3
[1,10)	14	12	2	15	12	3
[10,100)	9	8	1	11	8	3
[100,300)	1	1	0	3	1	2
Timeout	51	-	-	57	-	-

Table 6
Number of ILTP benchmark runs per runtime bin (in seconds) for $\mathsf{Seq}_{\mathsf{IPL}}$, split by prover and provability.

the tree-based prover had a mean runtime of 4.11 seconds and a median of 0.01 seconds. The proof sizes were also identical: the median was 152, the minimum was 2, the maximum was 11,532,210 and the mean was 189,242.

For $\mathsf{Seq}_{\mathsf{IPL}}$, we left out ten formulas from the benchmark set we used for $\mathsf{Seq}_{\mathsf{CPL}}$. For these formulas, the ILTP library does not specify whether they are valid in IPL, and the accompanying comments indicate that their status is unknown for existing IPL provers. GenZ also did not decide them.

We therefore tested 155 formulas for $\mathsf{Seq}_{\mathsf{IPL}}$, with average size 66 and median 55. Figure 7 shows the size distribution of the ILTP formulas used for $\mathsf{Seq}_{\mathsf{IPL}}$. For the zipper-based prover, the success rate was 67.74%, with a mean runtime of 98.09 seconds and a median of 0.14 seconds. The tree-based prover had a lower success rate of 63.87%, with a mean runtime of 112 seconds and a median of 0.25 seconds. Table 6 summarises the runtime histogram for the ILTP runs in $\mathsf{Seq}_{\mathsf{IPL}}$, using the same binning scheme employed in Table 5 for the LWB benchmarks. For both data structures, the completed proofs had a median size of 185, a minimum size of 2, a maximum size of 2,785,719, and a mean size of 102,257.

6. Conclusion and future work

We presented GenZ, a generic Haskell prover for sequent calculus. Our evaluation confirms that GenZ can reason fast on smaller modal formulas. The zipper-based implementation yields further improvements for calculi with loopchecks.

Future work will extend the scope of GenZ in two directions. First, we plan to study and implement proof search in the presence of analytic cut, thereby covering a wider family of logics [38, 39]. Second, we intend to extend our prover to other kinds of sequent-based formalisms, such as hypersequents or nested sequents. Preliminary work in this direction has been initiated in [40]. Integrating the focusing mechanism from [13] into GenZ to enable safe terminating strategies is an interesting direction for future work.

Acknowledgments

We thank SURF (www.surf.nl) for the support in using the National Supercomputer Snellius. This publication is part of the project "*PICON — Practical methods for Concept Interpolation in Realistic Ontologies*" with file number OCENW.M.23.448 of the research programme Open Competition Domain Science, which is partially financed by the Dutch Research Council (NWO) under the grant https://doi.org/10.61686/OTFBL33387. We also thank the Master of Logic programme at the Institute for Logic, Language and Computation, University of Amsterdam, for partially funding the first author's trip to present this work.

Declaration on Generative AI

During the preparation of this work, the first author used ChatGPT-4o for writing auxiliary bash scripts to run benchmark experiments and Claude Opus 4.6 for checking grammar and spelling, paraphrasing and rewording. After using these services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] G. Gentzen, Untersuchungen über das logische Schließen, Mathematische Zeitschrift 39 (1935) 176–210; 405–431. doi:10.1007/BF01201353.

[2] H. Horacek, Presenting proofs in a human-oriented way, in: Proceedings of CADE, volume 1632 of *LNCS*, Springer, 1999, pp. 142–156. doi:10.1007/3-540-48660-7_10.

[3] C. Alrabbaa, S. Borgwardt, A. Hirsch, N. Knieriemen, A. Kovtunova, A. M. Rothermel, F. Wiehr, In the head of the beholder: Comparing different proof representations, in: Proceedings of RuleML+RR, volume 13752, Springer, 2022, pp. 211–226. doi:10.1007/978-3-031-21541-4_14.

[4] C. Alrabbaa, F. Baader, S. Borgwardt, P. Koopmann, A. Kovtunova, Finding good proofs for description logic entailments using recursive quality measures, in: Proceedings of CADE, volume 12699, Springer, 2021, pp. 291–308. doi:10.1007/978-3-030-79876-5_17.

[5] J. Méndez, C. Alrabbaa, P. Koopmann, R. Langner, F. Baader, R. Dachselt, Evonne: A visual tool for explaining reasoning with OWL ontologies and supporting interactive debugging, Computer Graphics Forum 42 (2023). doi:10.1111/CGF.14730.

[6] R. Goré, J. Thomson, J. Wu, A history-based theorem prover for intuitionistic propositional logic using global caching: IntHistGC system description, in: Proceedings of IJCAR, volume 8562 of *LNCS*, Springer, 2014, pp. 262–268. doi:10.1007/978-3-319-08587-6_19.

[7] G. Corsi, G. Tassi, Intuitionistic logic freed of all metarules, The Journal of Symbolic Logic 72 (2007) 1204–1218. URL: http://www.jstor.org/stable/27588600.

[8] N. Olivetti, G. L. Pozzato, Nested sequent calculi and theorem proving for normal conditional logics: The theorem prover NESCOND, Intelligenza Artificiale 9 (2015) 109–125. doi:10.3233/IA-150082.

[9] M. Girlando, L. Straßburger, MOIN: A nested sequent theorem prover for intuitionistic modal logics(system description), in: Proceedings of IJCAR, Springer, 2020, pp. 398–407. doi:10.1007/978-3-030-51054-1_25.

[10] K. Brünnler, Deep sequent systems for modal logic, Archive for Mathematical Logic 48 (2009) 551–577. doi:10.1007/s00153-009-0137-3.

[11] V. Nigam, G. Reis, L. Lima, Quati: An automated tool for proving permutation lemmas, in: Proceedings of IJCAR, volume 8562 of *LNCS*, Springer, 2014, pp. 255–261.

[12] C. Olarte, E. Pimentel, C. Rocha, A rewriting logic approach to specification, proof-search, and meta-proofs in sequent systems, Journal of Logical and Algebraic Methods in Programming 130 (2023) 100827.

[13] V. Nigam, G. Reis, L. Lima, Towards the automated generation of focused proof systems, in: Proceedings of WoF, 2015, pp. 1–6. doi:10.4204/EPTCS.197.1.

[14] O. Gasquet, A. Herzig, D. Longin, M. Sahade, LoTREC: Logical tableaux research engineering companion, in: Proceedings of TABLEAUX, volume 3702 of *LNCS*, Springer, 2005, pp. 318–322. doi:10.1007/11554554_25.

[15] M. Ferrari, C. Fiorentini, G. Fiorino, JTabWb: A Java framework for implementing terminating sequent and tableau calculi, Fundamenta Informaticae 150 (2017) 119–142. doi:10.3233/FI-2017-1462.

[16] D. Tishkovsky, R. A. Schmidt, M. Khodadadi, MetTeL: A tableau prover with logic-independent inference engine, in: Proceedings of TABLEAUX, Springer, 2011, pp. 242–247.

[17] D. Tishkovsky, R. A. Schmidt, M. Khodadadi, MetTeL^2: Towards a tableau prover generation platform, in: Proceedings of PAAR, volume 21 of *EPiC Series in Computing*, EasyChair, 2012, pp. 149–162.

[18] J. Otten, The nanoCoP 2.0 connection provers for classical, intuitionistic and modal logics, in: Proceedings of TABLEAUX, Springer, 2021, pp. 236–249. doi:10.1007/978-3-030-86059-2_14.

[19] J. Brotherston, N. Gorogiannis, R. L. Petersen, A generic cyclic theorem prover, in: Proceedings of APLAS, Springer, 2012, pp. 350–367.

[20] Y. Zohar, A. Zamansky, Gen2sat: An automated tool for deciding derivability in analytic pure sequent calculi, in: Proceedings of IJCAR, Springer, 2016, p. 487–495. doi:10.1007/978-3-319-40229-1_33.

[21] G. Reis, Z. Naeem, M. Hashim, Sequoia: A playground for logicians: (system description), in: Proceedings of IJCAR, Springer, 2020, pp. 480–488.

[22] G. Huet, The zipper, Journal of functional programming 7 (1997) 549–554. doi:10.1017/S0956796897002864.

[23] T. van Laarhoven, MOLTAP — A modal logic tableau prover, 2009. URL: https://moltap.twanvl.nl/.

[24] P. Balsiger, A. Heuerding, S. Schwendimann, A benchmark method for the propositional modal logics K, KT, S4, Journal of Automated Reasoning 24 (2000) 297–317. doi:10.1023/A:1006249507577.

[25] T. Raths, J. Otten, C. Kreitz, The ILTP problem library for intuitionistic logic: Release v1. 1, Journal of Automated Reasoning 38 (2007) 261–271. doi:10.1007/s10817-006-9060-z.

[26] A. S. Troelstra, H. Schwichtenberg, Basic Proof Theory, Cambridge University Press, 2000.

[27] P. Blackburn, M. De Rijke, Y. Venema, Modal Logic, volume 53, Cambridge University Press, 2001.

[28] H. Wansing, Sequent systems for modal logics, in: Handbook of Philosophical Logic: Volume 8, Springer, 2002, pp. 61–145. doi:10.1007/978-94-010-0387-2_2.

[29] G. Sambin, S. Valentini, The modal logic of provability. the sequential approach, Journal of Philosophical Logic (1982) 311–342. URL: https://www.jstor.org/stable/30226252.

[30] R. Goré, R. Ramanayake, Valentini's cut-elimination for provability logic resolved, The Review of Symbolic Logic 5 (2012) 212–238.

[31] D. S. Shamkanov, Circular proofs for the Gödel-Löb provability logic, Mathematical Notes 96 (2014) 575–585. doi:10.1134/S0001434614090326.

[32] R. Dyckhoff, Intuitionistic decision procedures since Gentzen, Advances in proof theory (2016) 245–267. doi:10.1007/978-3-319-29198-7_6.

[33] M. Takano, Subformula property as a substitute for cut-elimination in modal propositional logics, Mathmatica japonica 37 (1992) 1129–1145.

[34] X. Yang, M. Gattinger, GenZ version 0.1.0.0, 2026. doi:10.5281/zenodo.20828698.

[35] C. Done, Weigh, 2023. URL: https://github.com/fpco/weigh, Haskell library.

[36] M. Kaminski, T. Tebbi, InKreSAT: Modal reasoning via incremental reduction to SAT, in: Proceedings of CADE, volume 7898 of *LNCS*, Springer, 2013, pp. 436–442. doi:10.1007/978-3-642-38574-2_31.

[37] F. Papacchini, C. Nalon, U. Hustadt, C. Dixon, Local is best: Efficient reductions to modal logic K, Journal of Automated Reasoning 66 (2022) 639–666. doi:10.1007/S10817-022-09630-6.

[38] A. Ciabattoni, T. Lang, R. Ramanayake, Cut-restriction: from cuts to analytic cuts, in: Proceedings of LICS, IEEE, 2023, pp. 1–13. doi:10.1109/LICS56636.2023.10175785.

[39] A. Ciabattoni, T. Lang, R. Ramanayake, Analytic proofs for tense logic, in: Proceedings of TABLEAUX, Springer, 2025, pp. 220–237. doi:10.1007/978-3-032-06085-3_12.

[40] S. Y. S. Chiu, Using zippers for nested sequents with focus, 2025. URL: https://eprints.illc.uva.nl/id/eprint/2391/1/MoL-2025-20.text.pdf, master thesis, University of Amsterdam.