MSc Thesis: Automated Proof Search in a Cyclic Proof System for Game Logic

Author: Christopher Worthington
First Supervisor: prof. dr. H.H. Hansen
Second Supervisor: dr. D.R.S. Ramanayake

June 2023

Abstract

This Master's thesis aims to implement a proof search algorithm that is sound and complete relative to the Game Logic sequent calculus, CloG. The results build upon another attempt to implement proof search for CloG in a Bachelor's thesis. In this previous thesis, an implementation was completed however there were limited concrete definitions and proofs provided. This project focuses on the theoretical definition of the algorithm by strictly defining the search space through a new altered proof system SCloG. An algorithm is implemented in the meta-programming language Rascal and is proven to be sound. An incomplete proof is given for the termination of the algorithm with a suggestion for how to prove the final condition. Partial results and observations are also given for the completeness of SCloG relative to CloG.

Contents

1 Introduction

Game Logic [1] is a relatively new branch of modal-logic introduced by Parikh in 1983 [2] to reason about two-player determined games. It branches from propositional dynamic logic (PDL) [3], which models the operations of a program. A modal operator in PDL then allows reasoning about the propositions that can be guaranteed after running a program. The execution of a program can also be seen as a person playing a single player determined game. Within Game Logic, there are two players in the game who are both making choices. From the perspective of each player there is non-determinism present based on what the other player may decide to do. There are two modal operators, which allow reasoning about what a player can force with regards to the end state of a given game while the other player is trying to prevent them from reaching their goal. If we name the players Angel and Demon then a modality in game logic of the form $\langle\gamma\rangle\varphi$ states that Angel has a strategy in the game γ to ensure φ holds in the final state. The second modality $[\gamma]\varphi$ states the same but that Demon can ensure φ instead.

An important aspect for any defined logic are formal proof systems. These formal proof systems give a syntactical definition for a derivable formula within a logical language. Once derivability is shown to be sound and complete relative to semantic validity, formal proof systems then provide a way to construct valid formulas in the language. Hilbert systems [4] are an early form of formal proof system with basic syntax and a sizeable list of axioms while only having a few rules. While Hilbert systems often have less complex syntax which is easier to read, they tend to not be suitable for proof search as there is no consistent strategy. Sequent calculus systems on the other hand, have more rules and limited axioms. Introduced by Gentzen, they are designed in such a way that reduces the search space for proof solving significantly. This is because going from the root upwards, only a limited number of rules can be applied per sequent excluding cut type rules due to how the rules are dependent on the form of the lower sequent. In addition, sequent calculus systems are usually only made up of introduction rules which make the proof analytic. The analytic property means that formulas are consistently broken down into smaller formulas of some kind going from the root upwards. Proof branches then can not infinitely extend and thus proof search will always terminate.

A Hilbert system is given by Parikh to define the derivability of a Game Logic formula. Two cyclic sequent calculus proof systems, G and CloG, are also defined for Game Logic in [5]. The paper uses these proof systems as intermediate transformation steps from Clo and CloM for the modal μ-calculus to the Hilbert System for Game Logic, which is referred to here as Par. These transformations are then used to prove completeness for the Hilbert system Par relative to those μ-calculus proof systems. As stated before, Hilbert Systems are not suited for proof search whereas sequent calculus systems are much more viable. This further motivates a transformation from the Game Logic sequent calculus proof systems to the Hilbert System Par, which is implemented in the two Bachelor theses [6, 7]. It is therefore desirable to find and implement a proof search algorithm for CloG which can then subsequently be used to find proofs in G and Par through the implemented transformation.

A Bachelor thesis by Meerholz [8] investigated and implemented several proof search strategies for CloG. The considerations for these strategies were whether they would be complete for CloG, and whether they would find the most optimal proofs in terms of size and redundant rule applications. While Meerholz's implementations are already useful to investigate CloG proof trees and search strategies, they cannot be used for a reliable proof search algorithm without concrete definitions and proofs. The aim of this project is to build on the previous investigation with a well defined algorithm about which properties can be stated and proven. The work towards proving soundness, completeness, and termination will then provide more insight in the viability of proof search for the system CloG. This then gives the following research question:

How can we define and implement a complete terminating algorithm for CloG?

A strategy will be selected from Meerholz's research which is expected to be complete relative to CloG while still finding the optimal solutions in terms of proof size in as many cases as possible. It will then be investigated whether this algorithm can be adjusted and then proven to be terminating and complete relative to CloG, while preserving soundness.

This thesis is structured with the following sections. Section 2 gives the preliminary definitions and notation for Game Logic and CloG that will be used throughout the thesis. A literature review is performed in Section 3 to investigate previous studies into proof search for both Game Logic and other cyclic sequent calculus systems. Section 4 covers the problems posed by proof search for CloG and showing termination and completeness, followed by the methodology for approaching these problems. Section 5 is where the algorithm is defined along with the soundness condition and pseudo-code for the implementation. In Section 6 a partial proof is given for termination of the algotithm. In Section 7, some proofs and lemmas are given that will lead to the completeness of the algorithm. Section 8 describes how the algorithm was implemented and the manual testing strategy used to validate it. Section 9 discusses the results of the project and the conclusions that can be made. Finally, Section 10 suggests what future developments can be made after this project for both CloG proof search and related subjects.

2 Game Logic Preliminaries

2.1 Language Definitions

Game Logic defines two sets of expressions, one for propositions which is denoted $\mathcal{L}_{\mathsf{GL}}$, and one for games which is denoted $\mathcal{G}_{\mathsf{GL}}$. Propositions use the set of propositional atoms denoted P_0 and games use the set of atomic games denoted G_0. Propositions have the usual operators of classical logic with the addition of the angelic $\langle\gamma\rangle$ and demonic $[\gamma]$ modalities where γ is a game. Games then have the dual operator ^{d}, concatenation ;, choice for angel $\sqcup$ or demon $\sqcap$, a test for angel ? or demon !, and finally angelic iteration $*$ and demonic iteration $^\times$.

Definition 2.1.

$$\begin{aligned}\mathcal{L}_{\mathsf{GL}} \ni \varphi \quad &::= \quad p \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle\gamma\rangle\varphi,\ \gamma \in \mathcal{G}_{\mathsf{GL}} \\ \mathcal{G}_{\mathsf{GL}} \ni \gamma \quad &::= \quad g \mid \gamma;\gamma \mid \gamma \sqcup \gamma \mid \gamma \sqcap \gamma \mid \gamma^* \mid \gamma^\times \mid \gamma^d \mid \gamma?,\ \varphi \in \mathcal{L}_{\mathsf{GL}}\end{aligned}$$

where $p \in \mathsf{P}_0$ and $g \in \mathsf{G}_0$.
The connectives $\rightarrow$, $\wedge$, $\leftrightarrow$ are the usual abbreviations from classical logic.
In $\mathcal{L}_{\mathsf{GL}}$: $[\gamma]\varphi = \neg\langle\gamma\rangle\neg\varphi$. In $\mathcal{G}_{\mathsf{GL}}$: $\gamma \sqcap \gamma = (\gamma^d \sqcup \gamma^d)^d$, $\gamma! = (\gamma^d?)^d$, and $\gamma^\times = (\gamma^{d^*})^d$

Due to the fact that the focus of this thesis is syntactical, detailed definitions for the semantics of Game Logic are not provided here. These details can be found in [5]. However, an intuitive description for the operators added by Game Logic is given.

Game Logic semantics are defined over a set of states and at each state there is a set of propositional atoms from P_0 that evaluate as true. As mentioned in Section 1, the modal operator $\langle\gamma\rangle\varphi$ states that Angel has a strategy in the game γ to ensure φ holds in the end state. Conversely, $[\gamma]\varphi$ states that Demon has a strategy in γ to ensure φ holds in the end state. A concatenation $\gamma;\delta$ is the game where γ is played followed by δ. For choice, $\gamma \sqcup \delta$ is the game where Angel picks whether γ or δ is played, while $\gamma \sqcap \delta$ is the game where Demon instead picks which game will be played. For a test, $\varphi?$ is the game where the proposition φ is evaluated in the current state and if it is false then Angel immediately loses, while $\varphi!$ is the game where Demon immediately loses if φ is true. For iteration, γ^* is the game where Angel can choose whether to play γ and then γ^* or stop the game before playing γ, while $\gamma^\times$ is the game where Demon chooses whether to play γ and then $\gamma^\times$ or stop. This creates iteration as in each case if the player chooses to play γ then they get the opportunity to choose again, and then as many times as they want to play before stopping. Lastly, the dual of a game γ^d is the game γ but with the roles of Angel and Demon reversed.

As an example, if the game $a^\times; p?; (b \sqcup c)^d$ is played then it has the following steps:

1. $a^\times$ is played.
   (a) Demon chooses whether to play a or not.
   (b) If yes then a is played followed by $a^\times$ again, otherwise continue.

2. $p?$ is played.
    (a) p is evaluated in the current state.
    (b) If p is false then Angel loses and the game ends, otherwise continue.
3. $(b \sqcup c)^d$ is played.
4. $b \sqcup c$ is played with roles swapped (equivalent to playing $b^d \sqcap b^d$).
    (a) Demon chooses whether to play b^d or c^d
    (b) Either b^d is played, or c^d is played

For the use in the later proof systems, negation normal forms are defined for $\mathcal{L}_{\mathsf{GL}}$ and $\mathcal{G}_{\mathsf{GL}}$. The normal forms restrict the use of negation $\neg$ and dual ^{d} only to atomic propositions and games respectively. The abbreviations $\psi \wedge \varphi$, $\gamma \sqcap \delta$, $\gamma!$, and $\gamma^\times$ are also used as primitives in the normal form language as they would otherwise not be expressible without negation and dual allowed on larger expressions.

Definition 2.2.

$$\begin{aligned} \mathcal{L}_{\mathsf{NF}} \ni \varphi \quad &::= \quad p \mid \neg p \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \langle \gamma \rangle \varphi, \ \gamma \in \mathcal{G}_{\mathsf{NF}} \\ \mathcal{G}_{\mathsf{NF}} \ni \gamma \quad &::= \quad g \mid g^d \mid \gamma ; \gamma \mid \gamma \sqcup \gamma \mid \gamma \sqcap \gamma \mid \gamma^* \mid \gamma^\times \mid \varphi? \mid \varphi!, \ \varphi \in \mathcal{L}_{\mathsf{NF}} \end{aligned}$$

where $p \in \mathsf{P}_0$ and $g \in \mathsf{G}_0$.

A set of "fixed points" is defined as the set of all formulas of the form $\langle \gamma^\circ \rangle \varphi$ where $\circ \in \{*, \times\}$. These fixed point formulas are important in the sequent calculus proof systems of Game Logic due to how they are unfolded in steps of the proof and lead to cycling branches. A distinction is also made between "least fixed points" of the form $\langle \gamma^* \rangle \varphi$ and "greatest fixed points" of the form $\langle \gamma^\times \rangle \varphi$. They are referred to as fixed points as the semantic valuation of these expressions in a Game Logic model involves the calculation of least and greatest fixed points respectively.

Definition 2.3. The set of *least fixed points* is denoted $F_* := \{\langle \gamma^* \rangle \varphi \mid \gamma \in \mathcal{G}_{\mathsf{NF}}, \varphi \in \mathcal{L}_{\mathsf{NF}}\}$. The set of *greatest fixed points* is denoted $F_\times := \{\langle \gamma^\times \rangle \varphi \mid \gamma \in \mathcal{G}_{\mathsf{NF}}, \varphi \in \mathcal{L}_{\mathsf{NF}}\}$. The full set of all fixed points is then denoted $F := F_* \cup F_\times$

A partial ordering is defined on the set of fixed points which uses the binary relation $\triangleleft$ for sub-formulas.

Definition 2.4. The partial order $\prec$ on F is defined such that $\langle \gamma^\circ \rangle \varphi \prec \langle \delta^\dagger \rangle \psi$ holds for $\circ, \dagger \in \{*, \times\}$ if $\delta^\dagger \triangleleft \gamma^\circ$. $\langle \gamma^\circ \rangle \varphi \preceq \langle \delta^\dagger \rangle \psi$ is additionally defined to hold if $\delta^\dagger \prec \gamma^\circ$ or $\delta^\dagger = \gamma^\circ$.

Some examples where this partial order holds and does not hold are as follows:

- $\langle a^\times \rangle p \preceq \langle (b \sqcup a^\times)^* \rangle q$ holds as $a^\times$ is a sub formula of $(b \sqcup a^\times)^*$ (). Whereas, $\langle (b \sqcup a^\times)^* \rangle p \preceq \langle a^\times \rangle q$ does not hold as $(b \sqcup a^\times)^*$ is not a sub formula of $a^\times$.
- $\langle a^\times \rangle p \preceq \langle a^\times \rangle q$ holds as $a^\times = a^\times$, but $\langle a^\times \rangle p \prec \langle a^\times \rangle q$ does not hold as $a^\times$ is not a sub formula of itself.
- $\langle a^\times \rangle p \preceq \langle b^\times \rangle q$ does not hold as the two fixed points are not comparable due to the fact that neither $a^\times \triangleleft b^\times$ nor $b^\times \triangleleft a^\times$ hold.

2.2 Proof Systems Par and G

Of the Game Logic proof systems discussed in [5], this project focuses on the system CloG as this is the system for which a proof search algorithm will be investigated. However, it is worth noting the differences in the systems and how this factors into the viability of proof search.

A proof in the Hilbert System, Par, consists of an ordered list of formulas. In this, each formula is either an axiom or a product of a rule applied to some previous formulas in the list The system Par has six axioms and three rules. It is stated that all tautologies of classical logic can be used as an axiom for Par and the rest contain arbitrary propositions φ, δ and arbitrary games γ, δ. See Figure 2.1 for the rules and axioms of Par and Figure 2.2 for an example proof. As mentioned in Section 1, the axioms provide infinitely many ways to start a proof, and also infinitely many ways to attempt working backwards from the goal. So a proof search in some search space for this system is not viable. A strategy for an automated proof solver may exist but this strategy is unclear and, as a transformation has been defined from the sequent calculus systems, a proof search for those is more worthwhile to investigate first.

Par **Axioms:**

1. All propositional tautologies.
2. $\langle\gamma;\delta\rangle\varphi \leftrightarrow \langle\gamma\rangle\langle\delta\rangle\varphi$
3. $\langle\gamma \sqcup \delta\rangle\varphi \leftrightarrow \langle\gamma\rangle\varphi \vee \langle\delta\rangle\varphi$
4. $\langle\gamma^*\rangle\varphi \leftrightarrow \varphi \vee \langle\gamma\rangle\langle\gamma^*\rangle\varphi$
5. $\langle\psi?\rangle\varphi \leftrightarrow \psi \wedge \varphi$
6. $\langle\gamma^d\rangle\varphi \leftrightarrow \neg\langle\gamma\rangle\neg\varphi$

Par **Rules:**

$$\frac{\varphi \qquad \varphi \to \psi}{\psi}\ \mathsf{MP}$$

$$\frac{\varphi \to \psi}{\langle\gamma\rangle\varphi \to \langle\gamma\rangle\psi}\ \mathsf{Mon}$$

$$\frac{\langle\gamma\rangle\varphi \implies \varphi}{\langle\gamma^*\rangle\varphi \implies \varphi}\ \mathsf{BarInd}$$

Figure 2.1: Axioms and rules of Par

1. $\langle a\rangle\neg p \vee \neg\langle a\rangle\neg p$ — Axiom 1) Tautology $A \vee \neg A$
2. $\langle a^d\rangle p \leftrightarrow \neg\langle a\rangle\neg p$ — Axiom 6) $\langle\gamma^d\rangle\varphi \leftrightarrow \neg\langle\gamma\rangle\neg\varphi$
3. $(\langle a^d\rangle p \leftrightarrow \neg\langle a\rangle\neg p) \to ((\langle a\rangle\neg p \vee \neg\langle a\rangle\neg p) \to (\langle a\rangle\neg p \vee \langle a^d\rangle p))$ — Axiom 1) Tautology $(A \leftrightarrow B) \to ((C \vee B) \to (C \vee A))$
4. $(\langle a\rangle\neg p \vee \neg\langle a\rangle\neg p) \to (\langle a\rangle\neg p \vee \langle a^d\rangle p)$ — Rule MP) 2, 3
5. $\langle a\rangle\neg p \vee \langle a^d\rangle p$ — Rule MP) 1, 4

Figure 2.2: Par proof for $\langle a\rangle\neg p \vee \langle a^d\rangle p$

G is a cut-free sequent calculus with deep inference rules. A sequent calculus proof is a tree where nodes are labelled by sets of formulas called a sequents. A sequent represents the disjunction of all formulas in the set, so for example, the sequent $\{\varphi, \psi, \chi\}$ is equivalent to $\varphi \vee \psi \vee \chi$. The sequents at the leaves of the tree must have the form of an axiom in the proof system. Rules then define how parent-child relationships are constructed between nodes in the tree dependent of the form of the sequents. The tree is presented with the root at the bottom going to the leaves at the top, and the rules reflect this. G-sequents are sets of Game Logic formulas from the normal form set $\mathcal{L}_{\mathsf{NF}}$. For both sequent calculus proof systems, saying a rule is "applied" to a sequent refers to when that rule is used with that sequent at the parent node. Then saying that a rule is applied to one or more of the formulas in that sequent, means that those are the active formulas in that rule application. The axiom Ax and the rules of G can be seen in Figure 2.3, where $\overline{\Phi}$ is the negation of Φ in the normal form $\mathcal{L}_{\mathsf{NF}}$. The leaves of the tree must all be instances of the axiom Ax. Given a well formed G proof tree, the root sequent is G provable in its disjunct form. An example proof in G is also shown in Figure 2.4.

$$\dfrac{}{\Phi,\overline{\Phi}}\,\mathsf{Ax} \qquad \dfrac{\Phi}{\Phi,\varphi}\,\mathsf{weak} \qquad \dfrac{\varphi,\psi}{\langle g\rangle\varphi,\langle g^d\rangle\psi}\,\mathsf{mod}_m \qquad \dfrac{\Phi,\varphi,\psi}{\Phi,\varphi\vee\psi}\,\vee \qquad \dfrac{\Phi,\varphi \qquad \Phi,\psi}{\Phi,\varphi\wedge\psi}\,\wedge$$

$$\dfrac{\Phi,\langle\gamma\rangle\varphi\vee\langle\delta\rangle\varphi}{\Phi,\langle\gamma\sqcup\delta\rangle\varphi}\,\sqcup \qquad \dfrac{\Phi,\langle\gamma\rangle\varphi\wedge\langle\delta\rangle\varphi}{\Phi,\langle\gamma\sqcap\delta\rangle\varphi}\,\sqcap \qquad \dfrac{\Phi,\psi\wedge\varphi}{\Phi,\langle\psi?\rangle\varphi}\,? \qquad \dfrac{\Phi,\psi\vee\varphi}{\Phi,\langle\psi!\rangle\varphi}\,!$$

$$\dfrac{\Phi,\psi(\gamma)}{\Phi,\psi(\chi!;\gamma)}\,\mathsf{mon}^{\mathsf{g}}_{\mathsf{d}} \qquad \dfrac{\Phi,\psi(\varphi)}{\Phi,\psi(\langle\chi!\rangle\varphi)}\,\mathsf{mon}^{\mathsf{f}}_{\mathsf{d}} \qquad \dfrac{\Phi,\psi(\langle\gamma\rangle\langle\delta\rangle\varphi)}{\Phi,\psi(\langle\gamma;\delta\rangle\varphi)}\,;_d$$

$$\dfrac{\Phi,\varphi\vee\langle\gamma\rangle\langle\gamma^*\rangle\varphi}{\Phi,\langle\gamma^*\rangle\varphi}\,* \qquad \dfrac{\Phi,\varphi\wedge\langle\gamma\rangle\langle\gamma^\times\rangle\varphi}{\Phi,\langle\gamma^\times\rangle\varphi}\,\times \qquad \dfrac{\Phi,\varphi\wedge\langle\gamma\rangle\langle(\overline{\Phi}!;\gamma)^\times\rangle\langle\overline{\Phi}!\rangle\varphi}{\Phi,\langle\gamma^\times\rangle\varphi}\,\mathsf{ind}_s$$

Figure 2.3: Axiom and rules of G

The deep inference rules $\mathsf{mon}^{\mathsf{g}}_{\mathsf{d}}$, $\mathsf{mon}^{\mathsf{f}}_{\mathsf{d}}$, and $;_d$ are rules that can be applied to sub-formulas of a formula ψ in a sequent, rather than only being applicable to the topmost operator. This opens up many more possible rule applications per sequent and therefore significantly increases the search space for proofs compared to a system without deep inference rules.

G also includes a *strengthened induction* rule ind_s which, going from the root upwards, embeds all other formulas in a sequent Φ into the unfolding of a greatest fixed point formula $\langle\gamma^\times\rangle\varphi$. This rule leads to possibly infinitely many different formulas that can appear in a sequent. This makes certain kinds of combinatorial analysis of proof trees for search strategies and proofs more difficult.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(\neg q\wedge\neg p), q\vee p}\,\mathsf{Ax}}{(\neg q\wedge\neg p),\langle q!\rangle p}\,!}{(\neg q\wedge\neg p),\langle q!\rangle p,\langle a^d\rangle\langle a^{d^*}\rangle\langle q!\rangle p}\,\mathsf{weak} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(\langle a^{\times}\rangle\langle\neg q?\rangle\neg p),\langle a^{d^*}\rangle\langle q!\rangle p}\,\mathsf{Ax}}{(\langle a^{\times}\rangle\langle\neg q?\rangle\neg p),(\neg q\wedge\neg p),\langle a^{d^*}\rangle\langle q!\rangle p}\,\mathsf{weak}}{(\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)\vee(\neg q\wedge\neg p),\langle a^{d^*}\rangle\langle q!\rangle p}\,\vee}{\langle(\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!\rangle(\neg q\wedge\neg p),\langle a^{d^*}\rangle\langle q!\rangle p}\,! \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\langle a^{\times}\rangle\langle\neg q?\rangle\neg p),\langle a^{d^*}\rangle\langle q!\rangle p}\,\mathsf{Ax}}{\langle a^{\times}\rangle\langle\neg q?\rangle\neg p),\langle a\rangle\langle(((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!;a)^{\times}\rangle\langle((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!\rangle(\neg q\wedge\neg p),\langle a^{d^*}\rangle\langle q!\rangle p}\,\mathsf{weak}}{\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)\vee\langle a\rangle\langle(((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!;a)^{\times}\rangle\langle((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!\rangle(\neg q\wedge\neg p),\langle a^{d^*}\rangle\langle q!\rangle p}\,\vee}{\langle\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!\rangle\langle a\rangle\langle(((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!;a)^{\times}\rangle\langle((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!\rangle(\neg q\wedge\neg p),\langle a^{d^*}\rangle\langle q!\rangle p}\,!}{\langle((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!\rangle(\neg q\wedge\neg p)\wedge\langle\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!\rangle\langle a\rangle\langle(((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!;a)^{\times}\rangle\langle((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!\rangle(\neg q\wedge\neg p),\langle a^{d^*}\rangle\langle q!\rangle p}\,\wedge}{\langle((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!\rangle(\neg q\wedge\neg p)\wedge\langle\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!;a\rangle\langle(((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!;a)^{\times}\rangle\langle((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!\rangle(\neg q\wedge\neg p),\langle a^{d^*}\rangle\langle q!\rangle p}\,;_d}{\langle(((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!;a)^{\times}\rangle\langle((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!\rangle(\neg q\wedge\neg p),\langle a^{d^*}\rangle\langle q!\rangle p}\,\times}{\langle a\rangle\langle(((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!;a)^{\times}\rangle\langle((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!\rangle(\neg q\wedge\neg p),\langle a^d\rangle\langle a^{d^*}\rangle\langle q!\rangle p}\,\mathsf{mod}_m}{\langle a\rangle\langle(((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!;a)^{\times}\rangle\langle((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!\rangle(\neg q\wedge\neg p),\langle q!\rangle p,\langle a^d\rangle\langle a^{d^*}\rangle\langle q!\rangle p}\,\mathsf{weak}}{(\neg q\wedge\neg p)\wedge\langle a\rangle\langle(((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!;a)^{\times}\rangle\langle((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!\rangle(\neg q\wedge\neg p),\langle q!\rangle p,\langle a^d\rangle\langle a^{d^*}\rangle\langle q!\rangle p}\,\wedge}{(\neg q\wedge\neg p)\wedge\langle a\rangle\langle(((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!;a)^{\times}\rangle\langle((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!\rangle(\neg q\wedge\neg p),\langle q!\rangle p\vee\langle a^d\rangle\langle a^{d^*}\rangle\langle q!\rangle p}\,\vee}{(\neg q\wedge\neg p)\wedge\langle a\rangle\langle(((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!;a)^{\times}\rangle\langle((\langle a^{\times}\rangle\langle\neg q?\rangle\neg p)!\rangle(\neg q\wedge\neg p),\langle a^{d^*}\rangle\langle q!\rangle p}\,*}{\langle a^{\times}\rangle(\neg q\wedge\neg p),\langle a^{d^*}\rangle\langle q!\rangle p}\,\mathsf{ind}_s$$

Figure 2.4: G proof for $\langle a^\times\rangle(\neg q\wedge\neg p),\langle a^{d^*}\rangle\langle q!\rangle p$ (equivalent to $\langle a^\times\rangle(\neg q\wedge\neg p)\vee\langle a^{d^*}\rangle\langle q!\rangle p$)

2.3 The Proof System CloG

CloG is another cut-free sequent calculus using formulas from $\mathcal{L}_{\mathsf{NF}}$ but, unlike G, it also uses an annotation system. Inspired by the μ-calculus proof system Clo in [9], formulas in CloG are annotated by a list of names. Each name x is introduced by the application of a new unfolding rule $\mathsf{clo}_{\mathsf{x}}$.

Definition 2.5. For each greatest fixed point formula $\varphi\in F_\times$ a countable set of names N_φ for unfoldings of φ is defined. The full set of names is denoted $N := \bigcup_{\varphi\in F_\times} N_\varphi$ and the individual names are typically denoted $\mathsf{x},\mathsf{x}_0,\mathsf{x}_1\ldots$. It is also assumed that for any fixed point formulas $\varphi,\psi\in F_\times$, it is the case that $N_\varphi\cap N_\psi=\emptyset$.

Names appear in annotations of formulas which take the form of sequences. The notation $s\in S$ is used throughout the paper for a sequence S if s occurs in S. Subset notation is also used for sequences such that for sequences S and S', $S\subseteq S'$ holds if all occurrences in S also occur in S'. The statement $S\subset S'$ also holds if $S\subseteq S'$ holds but not all occurrences in S' occur in S.

Definition 2.6. The set of *annotated formulas* is defined as $\mathcal{A}_N := \{\varphi^{\mathsf{a}} \mid \varphi \in \mathcal{L}_{\mathsf{NF}}, \mathsf{a} \in N^*\}$, where $*$ in this case is the kleene-star operation. Each CloG-Sequent is a set $\Phi \subset \mathcal{A}_N$ containing finitely many annotated formulas.

CloG proof trees are structured the same as in G but a CloG-sequent Φ is instead a set containing finitely many annotated formulas such that $\Phi \subset \mathcal{A}_N$. Each sequent is again a representation of the disjunction of all formulas in the set, but with annotations removed. So $\{(\varphi)^{\mathsf{a}}, (\psi)^{\mathsf{b}}, (\chi)^{\mathsf{c}}\}$ is equivalent to $\psi \vee \psi \vee \delta$. Branches can end in either an axiom or the "discharge" of the $\mathsf{clo_x}$ rule.

Some rules of CloG also include side conditions that restrict when rules can be applied beyond the direct form of a sequent. The partial order $\preceq$ on fixed points, formulated from Definition 2.4, is used in these conditions. Each name $\mathsf{x} \in N$ is associated with a fixed point formula $\varphi \in F_\times$ by being a member of the set n_φ. This association is used to extend the definition of $\preceq$ to the set of names N such that two names are comparable if their associated fixed points are too. Names can also be compared with fixed points using the same association. All of the fixed point unfolding rules then have a side condition that, in order to unfold an annotated fixed point φ^{a}, requires the active fixed point φ to be greater than all of the names in a according to the partial order $\preceq$.

Definition 2.7. Names inherit the order $\preceq$ on the set F of fixed point formulas. It is the case that $\forall \mathsf{x} \in N_\varphi$, $\forall \mathsf{y} \in N_\psi$, it is defined that $\mathsf{x} \preceq \mathsf{y}$ holds iff $\varphi \preceq \psi$. The ordering $\mathsf{x} \preceq \psi$ holds iff $\varphi \preceq \psi$. For a sequence of names $\mathsf{a} = \mathsf{x}_0, \mathsf{x}_1, ..., \mathsf{x}_{n-1} \in N^*$ and a fixed point formula $\varphi \in F$, $\mathsf{a} \preceq \varphi$ holds iff for all x_i occurring in a, $\mathsf{x}_i \in N_\varphi$ such that $\psi \preceq \varphi$.

The axiom $\mathsf{Ax1}$ and the full set of rules of CloG are shown in Figure 2.5. An example CloG proof is also given in Figure 2.6. The $\mathsf{clo_x}$ rule specifically includes an additional sequent of the form $[\Psi, \langle\gamma^\times\rangle\varphi^{\mathsf{ax}}]^{\mathsf{x}}$. This sequent is the "discharge" of the rule mentioned earlier, and can also close a branch along with the Ax rule. In a well formed CloG tree, any $\mathsf{clo_x}$ application must be accompanied by one or more discharge sequents within the sub proof tree rooted at that rule application. Then the root sequent of any well formed CloG tree is CloG derivable in its disjunct form as long as all annotations of the sequent are empty.

$$\frac{}{p^\epsilon, (\neg p)^\epsilon}\,\mathsf{Ax1} \qquad \frac{\varphi^{\mathsf{a}}, \psi^{\mathsf{b}}}{(\langle g\rangle\varphi)^{\mathsf{a}}, (\langle g^d\rangle\psi)^{\mathsf{b}}}\,\mathsf{mod}_m \qquad \frac{\Phi, \varphi^{\mathsf{a}}, \psi^{\mathsf{a}}}{\Phi, (\varphi \vee \psi)^{\mathsf{a}}}\,\vee \qquad \frac{\Phi, \varphi^{\mathsf{a}} \qquad \Phi, \psi^{\mathsf{a}}}{\Phi, (\varphi \wedge \psi)^{\mathsf{a}}}\,\wedge$$

$$\frac{\Phi, (\langle\gamma\rangle\varphi \vee \langle\delta\rangle\varphi)^{\mathsf{a}}}{\Phi, (\langle\gamma \sqcup \delta\rangle\varphi)^{\mathsf{a}}}\,\sqcup \qquad \frac{\Phi, (\langle\gamma\rangle\varphi \wedge \langle\delta\rangle\varphi)^{\mathsf{a}}}{\Phi, (\langle\gamma \sqcap \delta\rangle\varphi)^{\mathsf{a}}}\,\sqcap \qquad \frac{\Phi}{\Phi, (\varphi)^{\mathsf{a}}}\,\mathsf{weak} \qquad \frac{\Phi, (\varphi)^{\mathsf{ab}}}{\Phi, (\varphi)^{\mathsf{axb}}}\,\mathsf{exp} \qquad \frac{\Phi, (\langle\gamma\rangle\langle\delta\rangle\varphi)^{\mathsf{a}}}{\Phi, (\langle\gamma;\delta\rangle\varphi)^{\mathsf{a}}}\,;$$

$$(\mathsf{a} \preceq \langle\gamma^*\rangle\varphi)\,\frac{\Phi, (\varphi \vee \langle\gamma\rangle\langle\gamma^*\rangle\varphi)^{\mathsf{a}}}{\Phi, (\langle\gamma^*\rangle\varphi)^{\mathsf{a}}}\,* \qquad (\mathsf{a} \preceq \langle\gamma^\times\rangle\varphi)\,\frac{\Phi, (\varphi \wedge \langle\gamma\rangle\langle\gamma^\times\rangle\varphi)^{\mathsf{a}}}{\Phi, (\langle\gamma^\times\rangle\varphi)^{\mathsf{a}}}\,\times \qquad \frac{\Phi, (\psi \wedge \varphi)^{\mathsf{a}}}{\Phi, (\langle\psi?\rangle\varphi)^{\mathsf{a}}}\,? \qquad \frac{\Phi, (\psi \vee \varphi)^{\mathsf{a}}}{\Phi, (\langle\psi!\rangle\varphi)^{\mathsf{a}}}\,!$$

$$(\mathsf{a} \preceq \langle\gamma^\times\rangle\varphi, \mathsf{x} \in N_{\langle\gamma^\times\rangle\varphi}, \mathsf{x} \notin \Phi, \mathsf{a})\,\frac{\begin{array}{c}[\Phi, \langle\gamma^\times\rangle\varphi^{\mathsf{ax}}]^{\mathsf{x}}\\ \vdots \\ \Phi, (\varphi \wedge \langle\gamma\rangle\langle\gamma^\times\rangle\varphi)^{\mathsf{a}}\end{array}}{\Phi, (\langle\gamma^\times\rangle\varphi)^{\mathsf{a}}}\,\mathsf{clo_x}$$

Figure 2.5: Axiom and rules of CloG

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(\neg p)^{\epsilon}, p^{\epsilon}}\,\mathsf{Ax1}}{(\neg p)^{\mathsf{x}}, p^{\epsilon}}\,\mathsf{exp}}{(\neg p)^{\mathsf{x}}, p^{\epsilon}, (\langle a^{d}\rangle\langle a^{d^*}\rangle p)^{\epsilon}}\,\mathsf{weak} \qquad \dfrac{\dfrac{[(\langle a^{\times}\rangle\neg p)^{\mathsf{x}}, (\langle a^{d^*}\rangle p)^{\epsilon}]^{\mathsf{x}}}{(\langle a\rangle\langle a^{\times}\rangle\neg p)^{\mathsf{x}}, (\langle a^{d}\rangle\langle a^{d^*}\rangle p)^{\epsilon}}\,\mathsf{mod}_m}{(\langle a\rangle\langle a^{\times}\rangle\neg p)^{\mathsf{x}}, p^{\epsilon}, (\langle a^{d}\rangle\langle a^{d^*}\rangle p)^{\epsilon}}\,\mathsf{weak}}{(\neg p \wedge \langle a\rangle\langle a^{\times}\rangle\neg p)^{\mathsf{x}}, p^{\epsilon}, (\langle a^{d}\rangle\langle a^{d^*}\rangle p)^{\epsilon}}\,\wedge}{(\neg p \wedge \langle a\rangle\langle a^{\times}\rangle\neg p)^{\mathsf{x}}, (p \vee \langle a^{d}\rangle\langle a^{d^*}\rangle p)^{\epsilon}}\,\vee}{(\neg p \wedge \langle a\rangle\langle a^{\times}\rangle\neg p)^{\mathsf{x}}, (\langle a^{d^*}\rangle p)^{\epsilon}}\,*}{(\langle a^{\times}\rangle\neg p)^{\epsilon}, (\langle a^{d^*}\rangle p)^{\epsilon}}\,\mathsf{clo}_{\mathsf{x}}$$

Figure 2.6: CloG proof for $(\langle a^{\times}\rangle\neg p)^{\epsilon}, (\langle a^{d^*}\rangle p)^{\epsilon}$ (equivalent to $\langle a^{\times}\rangle\neg p \vee \langle a^{d^*}\rangle p$)

3 Literature Review

The proof system CloG is a cyclic sequent calculus. Cyclic proofs contain inductive theorem proving for fixed point logics, and in CloG this comes in the form of the clo_x rule. Each branch that ends in a sequent discharging a clo_x application can be read as a branch of infinite descent. The added complexity from a rule like clo_x comes from the addition of a global soundness check to detect the link from discharge sequents to the application of the clo_x rule further down the tree. Wehr [10] investigates these cyclic derivations in a general sense in his Master's thesis. The thesis shows how computationally costly some of these global soundness checks can be and attempts to abstract the content of cyclic derivations away from the logical content of the proof. The clo_x rule abstracts an infinitely extending branch with a closure condition which can be applied based on a global condition tracked by the annotations. The annotations reduce the need for an extensive soundness check on an entire branch but only on the current sequent and the sequent to which clo_x is applied. In this thesis, some further global conditions are used to detect redundant branches in a proof search, making use of the abstracted annotation system.

Marti and Venema [11] provide another study on cyclic proof systems. The study introduces a new cyclic proof system for a fragment of the modal μ-calculus. As stated in Section 1, Game Logic is a fragment of the μ-calculus. Game logic also has least and greatest fixed points as defined in Section 2. The new proof system introduced in the paper restricts the μ-calculus to the alternation-free fragment where least and greatest fixed points do not interact. These are called alternation-free fixed points. The system uses annotations like CloG but instead of a list of names for fixed points, the annotations simply indicate if a formula is "focused" or not. Rules are then restricted based on whether formulas are in focus or not. The use of alternation-free fixed points in this way make detecting infinitely unfolding fixed points much easier to find. The focus style annotations also provide a simpler way to track that it is the same fixed point being unfolded with the same rule applications for an inductive branch. Some observations are made in this thesis into the effects of limiting Game Logic to an even simpler *flat fragment* with no nested fixed points. However, this concept is not used in any of the completed proofs.

As mentioned in Section 1, Meerholz [8] has previously attempted to implement automated theorem proving for Game Logic in the proof system CloG. Proof search was executed by way of attempting different applicable rules, exploring possible incomplete proof trees until finding a well formed proof. It makes sense to build upon this implementation as there is already an investigation into the effectivity of different rule orderings and some base results for both soundness and completeness. These results are discussed in Section 4.3

Global soundness checks are avoided by Meerholz [8] by passing along a history of clo_x applications along the proof search so that the rest of the tree does not need to be continually queried. The search space is reduced not only by applying rules in a specific order but also by restricting certain rules to only be used in relation to other rules. This allows those rules to essentially be ignored in the proof search and can deterministically be applied dependent on other rule applications. In this project, a proof search algorithm will be developed from scratch, but many of the results from Meerholz [8] will be used to identify problems for proof search and decide the search strategy for the algorithm.

In [12], Brotherston, Gorogiannis and Petersen introduce a general framework, "CYCLIST", for automated theorem proving in cyclic proof systems. In the development of the framework, different strategies of proof search were attempted, in which inference rules are tried in different orders and at each point performing soundness checking for a well formed CloG proof. The implementation of CYCLIST is held back by the fact that the framework is made for cyclic proof systems which are cut-free, and it is certain that cut elimination does not hold for all cyclic proof systems.

CloG is cut-free so CYCLIST could theoretically be applied, but CYCLIST is a general implementation without the possibility of specific search strategies. This thesis is building on the results by Meerholz [8], so it is desirable to utilise the specific search strategies for CloG, making CYCLIST not a good solution in this case. However, some results are still usable. Both breadth and depth first search strategies were investigated for proof search in CYCLIST. Brotherston, Gorogiannis and Petersen [12] observe that breadth first search generally leads to smaller proofs when one is found, bur the depth first search strategy is much more practical due to the wide fan-out degree of the search space. An depth first search will generally find an existing solution much faster, and this is more important than the solution being optimal in size. It was also found that the specific rule orderings attempted had an effect on performance and the shape of proofs found.

4 Problem and Approach

4.1 Problem Statement

The aim of this project is to design a proof search algorithm for the Game Logic sequent calculus, CloG. The algorithm must only find well formed CloG proofs, thus being sound. The algorithm must also be defined in a such a way that concrete mathematical statements can be made about the algorithm, and these statements can be proven. For any root CloG-sequent the algorithm should be proven to always terminate, both in the case of an existing proof and no existing proof. If possible, it should also be proven that given a CloG proof exists for a root CloG-sequent, then the algorithm will find a proof for that same root sequent, making the algorithm complete relative to CloG.

The algorithm must be accompanied by an implementation which will take a CloG-Sequent as input and will either output a well formed CloG proof for the sequent or report that no proof exists. The implementation should define abstract data types for Game Logic and CloG. It should be possible to input both types via some concrete syntax and also have a readable form of output. Effort should also be made to ensure that the implementation is at a reasonable level of efficiency for a proof of concept, but the focus of this project will not be on optimisation. Pseudo-code should also be provided to support the link between the concrete theoretical algorithm definition and the implementations.

4.2 Problems for Proof Search in CloG

A mentioned in Section 3, Meerholz's Bachelor thesis [8] investigated several challenges in developing a proof solver, specifically using a proof search strategy, for CloG. The first is that the structural rules weak and exp are not dependent on a specific form of proposition in a given sequent. This means that for any breadth or depth first proof search strategy, they will add to the possible rules applicable at any sequent and increase the branches necessary to explore significantly.

Both the $\mathsf{clo_x}$ and $\times$ rules perform greatest fixed point unfoldings but are differentiated by whether the unfolding will be discharged at the end of a branch above in the proof. This poses a problem for a search strategy from the root as it cannot be decided which should be applied. Meerholz [8] suggests that the $\times$ rule can be excluded from proof search until it is found that a solution exists, at which point any $\mathsf{clo_x}$ applications from the search without a matching discharge can be converted to applications $\times$ in order to obtain the CloG proof. The found solution where some applications of $\mathsf{clo_x}$ may not be discharged will be referred to as a *partial* CloG proof.

As discussed in Section 1, a key property of non-cyclic sequent calculi for a terminating search strategy is the analytic reduction of formula complexity by each rule in the system. CloG has the unfolding rules $*$, $\mathsf{clo_x}$ and $\times$ which pose a problem for this termination condition. The unfolding rules make it possible for a infinitely extending branch to form in a CloG proof tree. This is because of the fact that when unfolding a fixed point formula φ, a more complex formula is added above the rule application containing φ as a sub-formula. If the more complex formula can be broken down to φ, then φ can be unfolded again. The $\mathsf{clo_x}$ rule is designed to detect a repeating unfolding and close a branch in this case, but not all cases of a cycling branch lead to a correct discharge and applications of $*$ cannot be discharged.

Any terminating search strategy for CloG will need to include a cycle detection condition to ensure that no infinitely extending branches occur in a proof due to the unfolding rules. In addition to this, no possible CloG proofs should missed by stopping the cycles from occurring. One option is to not allow unfoldings of the same formula in the same sequent, as the sub-proof above the later unfolding higher in the proof tree can be substituted to come after the first unfolding instead. However, while there are finitely many possible formulas that can appear in the sequents of one CloG proof branch, each $\mathsf{clo_x}$ application introduces a new label on a branch. This means that as long as applications of $\mathsf{clo_x}$ are not limited, there are infinitely many possible sequents. So the same fixed point can still unfold infinitely many times on a branch in always different sequents. The strategy must therefore be able to detect and halt infinitely repeating applications of $\mathsf{clo_x}$ on a branch that will not lead to new CloG proofs. Finitely many $\mathsf{clo_x}$ applications on a branch will ensure that there are finitely many names and subsequently finitely many possible sequents.

4.3 Strategy for Rule Application Ordering

Meerholz [8] denotes *simple rules* as all of the rules in CloG that depend on only one formula in the sequent to which they are applied and which have no global soundness conditions. This property applies to all rules besides. When only the simple rules are applied, they create isolated chains of formulas in a proof branch that are analytically deconstructed without interacting with the other formulas in the sequent. Due to this isolation, the order in which the rules are applied within proof segments of only simple rules has little effect on the size of a proof and will never affect whether the full CloG proof is well formed or not. If one simple rule can be applied to multiple formulas in the same sequent, the order in which the rule is applied to those formulas will not affect finding a CloG proof or not. The only one of these simple rules which has a consistent effect on proof size is the $\wedge$ rule as it will split the CloG proof into two branches. If there are simple rules that could be applied before $\wedge$ but are applied after, then they will be applied on both sides of the tree, creating redundancy. So it makes sense to apply $\wedge$ as late as possible amongst the simple rules. Other than $\wedge$, the simple rules can be applied in any arbitrary order.

Therefore, when considering possible rules to apply to a sequent, each of the simple rules can be given an arbitrary priority, with $\wedge$ having the lowest priority. Each rule will be considered to not be applicable as long as higher priority rules are still applicable. Then if no CloG proof can be found after applying a simple rule to one formula in a sequent, proof search can be be considered failed as applying the simple rule later will not affect the outcome.

Meerholz presents four different strategies that each compare applying clo_x and mod_m either before or after saturation of the current sequent. This gives insight into the best approach for when to apply clo_x and mod_m in relation to the other rules. When clo_x is applied as early as possible, this usually results in smaller proofs if they are found. This is because when clo_x is applied later where it could have been applied earlier, this often requires additional rule applications where the branch could have been discharged if clo_x was applied earlier. However, Meerholz [8] was only able to show that applying clo_x as late as possible to results in a complete algorithm by showing that it can always be swapped with the rule above it. The decision in made for the algorithm in this thesis to try applying clo_x at any time it is applicable, without stopping the proof search if no proof is found from a clo_x application. This way, the simpler proofs will be found if they exist, but the algorithm will remain complete with respect to CloG as clo_x will be applied later if necessary.

The mod_m rule cannot be applied unless there are no other formulas besides the pair of active formulas. However, if a sequent contains two annotated formulas of the form $(\langle a\rangle\varphi)^\mathsf{a}$ and $(\langle a^d\rangle\psi)^\mathsf{b}$, then the weak rule can be applied to all of the other annotated formulas and then mod_m can be applied. The mod_m is therefore considered to be applicable to a sequent containing formulas of the form mentioned, assuming that the other formulas will always be removed by weak in the same way. If mod_m can be applied to a given pair of formulas and a CloG proof can be found in that case, then applying it to that same pair earlier or later will have no effect on finding a proof because the sequent above mod_m is always the same.

Unlike the simple rules, when there are multiple available pairs to which mod_m is applicable, then mod_m is not applicable to either pair after applying mod_m since the pair will be removed from the sequent. The mod_m rule will also affect whether any other CloG rules can be applied, so proof search should not be considered failed if applying mod_m does not result in a CloG proof, and other rule applications have not yet been attempted. It therefore would be easiest for the implementation of the rule ordering, to apply mod_m at the lowest priority so that all other possible rules must have been applied before it and no solutions will be lost by applying mod_m.

4.4 Approach for Algorithm Formulation

In Section 3, it was decided that the most efficient proof solving approach for this type of system is to implement a depth first search on applicable rules to each sequent. With this methodology, small increases in the complexity of the root sequent can massively increase the number of applicable rules and the orders in which they can be applied. It will therefore be necessary to reduce the search space significantly in order for the algorithm implementation to be viable. Ways to reduce the search space have been discussed in the previous sections but a structured way to impose the restrictions is required.

For this purpose, the search space for finding a proof for a given root sequent will be defined as a search tree. The construction of the search tree will be deterministic, and each branch of the tree will represent one possible ordering of rule applications. A deterministic algorithm can then be defined to only search this search tree rather than all possibilities given by CloG. Stricter conditions can be placed on the rules that can be applied at each sequent in the search space, such as the rule priority. Rules can also be added or altered into *search rules* such that they represent a combination of CloG rule applications. It is then necessary that those search rule applications can be converted back into well formed CloG proof segments via a transformation. An example of this would be a mod_m search rule that can be applied to a sequent with arbitrary side formulas, which is then transformed into applications of the CloG weak rule that removes the side formulas before the application of CloG mod_m.

The strategy described in Section 4.3 is not deterministic for the order in which rules are applied when the same rule can be applied to multiple formulas in a sequent. An order on the formulas in a sequent will therefore need to be kept within the search tree in order to also have a priority on the formulas in one sequent. So sequents in the tree will be sequences without repeats. This does not cause an issue when transforming to CloG as the CloG sequent will simply be the set of elements in the search tree sequence.

Another benefit of the search tree definition is that statements can be shown and proven about the search space that then can also be stated about the algorithm. A definition of a derivation in the search tree will be given in the form of a sub-tree with one of the possible orderings of rules. A transformation will be defined from one of these search derivations to a well formed CloG proof tree. The algorithm will be defined to terminate when finding a search derivation sub-tree and output the result of the transformation of the search derivation into a CloG proof. It is then certain that the algorithm will only output well formed CloG proofs and is therefore sound.

In order to ensure termination, conditions will be placed on each rule to stop a infinitely repeating cycle from occurring. Then it will be proven that the search tree can only be finite, and hence, the algorithm must terminate whether it finds a derivation or not. For completeness relative to CloG, it must be shown that if a proof exists in CloG for a root CloG-sequent, then the search tree for the same root sequent contains at least one derivation.

4.5 Detecting Cycles

As previously discussed, the rule that can lead to an infinitely cycling branch is the clo_x rule. This is due to the fact that the rule adds a new name x whenever it is applied and results in infinitely many possible sequents. The unfolding rules can be restricted to only be applicable to the same annotated fixed point formula in a given sequent once per branch. Although this will not result in certain termination unless names are limited. There are finitely many sets of formulas without annotations that can appear in a sequent of one proof. Therefore in a branch with infinitely many applications of clo_x, there must be cases in which clo_x is applied to the same fixed point formula with the same formulas (with annotations removed) in the sequent. When there are multiple applications of clo_x to the same fixed point formulas with the same formulas without annotations in the sequent on a branch, these will be referred to as *matching* clo_x applications. If there are matching applications clo_x and clo_y where clo_y is further up the tree, there is only one case in which the branch would not be discharged. Namely, when the annotations of the formulas in the later sequent are not super-sets of the annotations on the same formulas from the clo_x unfolding.

The example in Figure 4.1 shows an example of these matching $\mathsf{clo}_{\mathsf{x}}$ applications occurring. A name is introduced by the application of $\mathsf{clo}_{\mathsf{x}_0}$ to $(\langle c^{\times}\rangle p)^{\epsilon}$ and then later recorded in another unfolding with name x_1. The branch returns again to the same set of formulas that were unfolded by $\mathsf{clo}_{\mathsf{x}_1}$, but x_0 is no longer present. This is able to repeat again in the same way with x_2 and x_3 and can then continue to cycle infinitely.

$$
\begin{prooftree}
\AxiomC{$\vdots$}
\AxiomC{$\vdots$}
\AxiomC{$\vdots$}
\AxiomC{$\vdots$}
\AxiomC{$\vdots$}
\UnaryInfC{$(\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (\langle b^{\times}\rangle q)^{\mathsf{x}_1,\mathsf{x}_3}$}
\RightLabel{mod_m}
\UnaryInfC{$(\langle b^{d}\rangle\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (\langle b\rangle\langle b^{\times}\rangle q)^{\mathsf{x}_1,\mathsf{x}_3}$}
\RightLabel{weak}
\UnaryInfC{$(\langle c\rangle\langle c^{\times}\rangle p)^{\mathsf{x}_2}, (\langle b^{d}\rangle\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (\langle b\rangle\langle b^{\times}\rangle q)^{\mathsf{x}_1,\mathsf{x}_3}$}
\RightLabel{$\wedge$}
\BinaryInfC{$(\langle c\rangle\langle c^{\times}\rangle p)^{\mathsf{x}_2}, (\langle b^{d}\rangle\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (q \wedge \langle b\rangle\langle b^{\times}\rangle q)^{\mathsf{x}_1,\mathsf{x}_3}$}
\RightLabel{$\mathsf{clo}_{\mathsf{x}_3}$}
\UnaryInfC{$(\langle c\rangle\langle c^{\times}\rangle p)^{\mathsf{x}_2}, (\langle b^{d}\rangle\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (\langle b^{\times}\rangle q)^{\mathsf{x}_1}$}
\RightLabel{$\wedge$}
\BinaryInfC{$(p \wedge \langle c\rangle\langle c^{\times}\rangle p)^{\mathsf{x}_2}, (\langle b^{d}\rangle\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (\langle b^{\times}\rangle q)^{\mathsf{x}_1}$}
\RightLabel{$\mathsf{clo}_{\mathsf{x}_2}$}
\UnaryInfC{$(\langle c^{\times}\rangle p)^{\epsilon}, (\langle b^{d}\rangle\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (\langle b^{\times}\rangle q)^{\mathsf{x}_1}$}
\RightLabel{$\vee$}
\UnaryInfC{$(\langle c^{\times}\rangle p \vee \langle b^{d}\rangle\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (\langle b^{\times}\rangle q)^{\mathsf{x}_1}$}
\RightLabel{$*$}
\UnaryInfC{$(\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (\langle b^{\times}\rangle q)^{\mathsf{x}_1}$}
\RightLabel{mod_m}
\UnaryInfC{$(\langle b^{d}\rangle\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (\langle b\rangle\langle b^{\times}\rangle q)^{\mathsf{x}_1}$}
\RightLabel{weak}
\UnaryInfC{$(\langle c\rangle\langle c^{\times}\rangle p)^{\mathsf{x}_0}, (\langle b^{d}\rangle\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (\langle b\rangle\langle b^{\times}\rangle q)^{\mathsf{x}_1}$}
\RightLabel{$\wedge$}
\BinaryInfC{$(\langle c\rangle\langle c^{\times}\rangle p)^{\mathsf{x}_0}, (\langle b^{d}\rangle\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (q \wedge \langle b\rangle\langle b^{\times}\rangle q)^{\mathsf{x}_1}$}
\RightLabel{$\mathsf{clo}_{\mathsf{x}_1}$}
\UnaryInfC{$(\langle c\rangle\langle c^{\times}\rangle p)^{\mathsf{x}_0}, (\langle b^{d}\rangle\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (\langle b^{\times}\rangle q)^{\epsilon}$}
\RightLabel{$\wedge$}
\BinaryInfC{$(p \wedge \langle c\rangle\langle c^{\times}\rangle p)^{\mathsf{x}_0}, (\langle b^{d}\rangle\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (\langle b^{\times}\rangle q)^{\epsilon}$}
\RightLabel{$\mathsf{clo}_{\mathsf{x}_0}$}
\UnaryInfC{$(\langle c^{\times}\rangle p)^{\epsilon}, (\langle b^{d}\rangle\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (\langle b^{\times}\rangle q)^{\epsilon}$}
\RightLabel{$\vee$}
\UnaryInfC{$(\langle c^{\times}\rangle p \vee \langle b^{d}\rangle\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (\langle b^{\times}\rangle q)^{\epsilon}$}
\RightLabel{$*$}
\UnaryInfC{$(\langle b^{d^*}\rangle\langle c^{\times}\rangle p)^{\epsilon}, (\langle b^{\times}\rangle q)^{\epsilon}$}
\end{prooftree}
$$

Figure 4.1: Infinitely extending CloG branch from repeated applications of clo

It is not the aim to identify every branch and sequent in which this occurs, but rather consider the cases where this repetition will not lead to any new possible CloG proof. See the following generalisation of the structure of a branch containing matching applications $\mathsf{clo}_{\mathsf{x}_0}$, $\mathsf{clo}_{\mathsf{x}_1}$, and $\mathsf{clo}_{\mathsf{x}_2}$:

$$
\begin{prooftree}
\AxiomC{$\vdots$}
\RightLabel{$\mathsf{clo}_{\mathsf{x}_2}$}
\UnaryInfC{$\Phi_3, (\langle\gamma\rangle\varphi)^{\mathsf{c}}$}
\noLine
\UnaryInfC{$\vdots$ (segment 2)}
\RightLabel{$\mathsf{clo}_{\mathsf{x}_1}$}
\UnaryInfC{$\Phi_2, (\langle\gamma\rangle\varphi)^{\mathsf{b}}$}
\noLine
\UnaryInfC{$\vdots$ (segment 1)}
\RightLabel{$\mathsf{clo}_{\mathsf{x}_0}$}
\UnaryInfC{$\Phi_1, (\langle\gamma\rangle\varphi)^{\mathsf{a}}$}
\end{prooftree}
$$

As stated before, it is the case that the only difference between matching applications is in the annotations, since they would otherwise be discharged. The possible CloG proofs that these different annotations may lead to after the application of $\mathsf{clo}_{\mathsf{x}_2}$ are dependent on the $\mathsf{clo}_{\mathsf{x}}$ applications in segment 2. It is suspected that in an infinitely extending branch there will be cases where the annotations introduced by $\mathsf{clo}_{\mathsf{x}}$ applications in segment 2 are equivalent in some way to those in segment 1 so that they do not lead to any different CloG proofs. That would allow for the proof search to be stopped at the sequent $\Phi_3, (\langle\gamma\rangle\varphi)^{\mathsf{c}}$. A working definition has not yet been found to identify when the change in annotations is redundant for the proof search. An additional condition will still be added in Section 5 halt the proof search in that situation, but the definition of the condition will for now be left arbitrary. A proof that a condition of this kind leads to termination is given in Section 6 with the support of an assumption that a cycling unfolding will occur.

4.6 Approach for Implementation

For the algorithm implementation, the search tree will not be constructed in its entirety, but will be searched using a recursive proof finding function. This will implicitly form a tree and will stop either when there are no more applicable rules or when a successful branch is found. The transformation from a successful search branch to CloG will also be carried out when returning back up the recursive tree, rather than returning the entire search branch and transforming after. By utilising recursion like this, the need for a data representation of the search tree is not necessary beyond storing the order of the formulas in each sequent until transforming to a CloG sequent. Lists will be used for this with removal of duplicates at each rule application to ensure that repeats do not occur. Abstract data types for Game Logic and CloG are already defined in the meta-programming language Rascal from the previous Bachelor Theses [8, 6, 7], so it follows to use Rascal again for this project to avoid re-implementing the base of the project.

5 Algorithm Definition and Soundness

5.1 Definition of SCloG

SCloG is a proof search system for game logic, which is derived from the rules of CloG and if the search is successful, will result in finding a CloG proof. An SCloG search tree can be seen as a reduced search space of possible CloG proofs, which can be exhaustively searched in order to find one or no proof. An SCloG derivation will also be defined as a sub-tree of an SCloG search tree, from which a CloG proof can be obtained via transformation. The definition starts with what an SCloG sequent is in contrast to a CloG-sequent. A CloG-sequent is any set of annotated formulas but, in order to put more constraints on the order in which rules are applied, SCloG uses sequences of formulas.

Definition 5.1. An *SCloG-sequent*, Γ, is a non-repeating non-empty sequence of annotated formulas $\varphi_0^{\mathsf{a}_0}$; ... ; $\varphi_n^{\mathsf{a}_n}$. The set of all SCloG-sequents is $\mathcal{S} := \{\Gamma \mid \Gamma \in {\mathcal{A}_N}^+,\ \forall \varphi_n^{\mathsf{a}_n}, \varphi_m^{\mathsf{a}_m} \in \Gamma : \varphi_n^{\mathsf{a}_n} = \varphi_m^{\mathsf{a}_m} \to m = n\}$ where $+$ is the kleene-plus operation.

Definition 5.2. The set form of an SCloG-sequent Γ is denoted $\Phi_\Gamma^N := \{\varphi^{\mathsf{a}} \mid \varphi^{\mathsf{a}} \in \Gamma\}$. The set form of an SCloG-sequent Γ with no annotations is denoted $\Phi_\Gamma := \{\varphi \mid \varphi^{\mathsf{a}} \in \Gamma\}$.

When applying a rule to an SCloG sequent Γ, this may introduce a new annotated formula that already exists in Γ. This would introduce a repeat due to SCloG sequents being sequences and not sets. So a duplicate removal function is defined in order to later solve this problem once the rules are defined. It preserves the first occurrence of each element as the algorithm will be written in Rascal which implements its duplicate removal function for lists in the same way.

Definition 5.3. The function $dup : {\mathcal{A}_N}^* \to \mathcal{S}$ removes all duplicate values from a sequence of annotated formulas. The first occurrence of each annotated formula remains in the sequence.

In an SCloG search tree, each vertex represents a sequent of the proof. The tree will be built upwards from the root at the bottom to the leaves at the top, with each vertex having some number of edges going outwardly from it. Each of the edges will represent a possible rule that can be applied at the sequent associated with the vertex. In a CloG proof, there are some side conditions on fixed point unfolding rules that rely on information about unfolded fixed point formulas further up the tree, so local decisions cannot be made on vertices without additional recorded information. The need to discharge the clo rule at the end of the branch requires information about entire CloG-sequents further down the tree. For this, some structures are defined that store information about a specific fixed point unfolding, which record both the annotated fixed point formula that was unfolded, and the set of annotated formulas in the sequent at that point.

Definition 5.4. The set of *least fixed point unfolding recordings* is denoted $U_* := ((F_* \times N^*) \times \mathcal{P}(\mathcal{A}_N))$. The set of *greatest fixed point unfolding recordings* is denoted $U_\times := ((F_\times \times N^*) \times \mathcal{P}(\mathcal{A}_N))$. The full set of *fixed point unfolding recordings* is denoted $U := U_* \cup U_\times$.

The example CloG proof segment in Figure 5.1 contains applications of clo_x and $*$. A greatest fixed point unfolding recording for the application of clo_{x_2} would have the form $((\langle a^{d^\times}\rangle\langle a^{d^{\times\times}}\rangle\neg p)^{\mathsf{x}_0}, \{(\langle a^*\rangle\langle a^{**}\rangle p)^{\varepsilon}, (\langle a^{d^\times}\rangle\langle a^{d^{\times\times}}\rangle\neg p)^{\mathsf{x}_0}\})$ while the upper application of $*$ would have a least fixed point unfolding recording of the form $((\langle a^{**}\rangle p)^{\varepsilon}, \{(\langle a^{**}\rangle p)^{\varepsilon}, (\langle a\rangle\langle a^*\rangle\langle a^{**}\rangle p)^{\varepsilon}, (\langle a^{d^{\times\times}}\rangle\neg p \land \langle a^d\rangle\langle a^{d^\times}\rangle\langle a^{d^{\times\times}}\rangle\neg p)^{\mathsf{x}_0,\mathsf{x}_2}\})$

$$\dfrac{\dfrac{\dfrac{\dfrac{(p \lor \langle a^*\rangle\langle a^{**}\rangle p)^{\varepsilon}, (\langle a\rangle\langle a^*\rangle\langle a^{**}\rangle p)^{\varepsilon}, (\langle a^{d^{\times\times}}\rangle\neg p \land \langle a^d\rangle\langle a^{d^\times}\rangle\langle a^{d^{\times\times}}\rangle\neg p)^{\mathsf{x}_0,\mathsf{x}_2}}{(\langle a^{**}\rangle p)^{\varepsilon}, (\langle a\rangle\langle a^*\rangle\langle a^{**}\rangle p)^{\varepsilon}, (\langle a^{d^{\times\times}}\rangle\neg p \land \langle a^d\rangle\langle a^{d^\times}\rangle\langle a^{d^{\times\times}}\rangle\neg p)^{\mathsf{x}_0,\mathsf{x}_2}}\,*}{(\langle a^{**}\rangle p \lor \langle a\rangle\langle a^*\rangle\langle a^{**}\rangle p)^{\varepsilon}, (\langle a^{d^{\times\times}}\rangle\neg p \land \langle a^d\rangle\langle a^{d^\times}\rangle\langle a^{d^{\times\times}}\rangle\neg p)^{\mathsf{x}_0,\mathsf{x}_2}}\,\lor}{(\langle a^*\rangle\langle a^{**}\rangle p)^{\varepsilon}, (\langle a^{d^{\times\times}}\rangle\neg p \land \langle a^d\rangle\langle a^{d^\times}\rangle\langle a^{d^{\times\times}}\rangle\neg p)^{\mathsf{x}_0,\mathsf{x}_2}}\,*}{(\langle a^*\rangle\langle a^{**}\rangle p)^{\varepsilon}, (\langle a^{d^\times}\rangle\langle a^{d^{\times\times}}\rangle\neg p)^{\mathsf{x}_0}}\,\mathsf{clo}_{\mathsf{x}_2}$$

Figure 5.1: Proof segment containing applications of clo_x and $*$

A labelling function is used to associate SCloG sequents and SCloG rules with vertices and edges respectively. The only relevant greatest fixed point unfoldings for a given sequent are those associated with the names in the annotated formulas within the sequent. Therefore, a map from names to elements of $U_\times$ is used to record and lookup those greatest fixed point unfoldings in each vertex label. A subset of U_* is also kept in each vertex label, as preventing multiple applications of $*$ at the same sequent is an essential step in ensuring termination. The set of names that are already generated on the branch is also stored in each sequent node label. There is one exception to vertices that represent the point at which a CloG proof splits into two proof branches. An additional vertex type $\wedge$ is added for this that has two outgoing branches representing the possible left and right branches of the CloG proof stemming from the CloG $\wedge$ rule.

Definition 5.5. An *SCloG search tree* structure is a labelled tree of the form $T := (V, E, l_V, l_E)$. Where, V is a set of vertices, $E \subseteq V \times V$ is a set of edges such that (V,E) is a tree, and both $l_V : V \to L_V$ and $l_E : E \to L_E$ are labelling functions.
The vertex label set L_V is of the form $L_V = L_{\mathcal{S}} \cup \{\wedge\}$ where elements of $L_{\mathcal{S}}$ are of the form $(N_V, \mathfrak{h}, \mathcal{H}, \Gamma)$ where:

- $N_V \in N^*$, a sequence containing all names introduced on the branch ending in a vertex $v \in V$, in the order they were introduced per clo rule.
- $\mathfrak{h} : N \rightharpoonup U_\times$, an injective partial function, which maps a name y to the fixed point unfolding that introduced y on the branch ending in v
- $\mathcal{H} \subseteq U_*$, a set of fixed point unfolding recordings that stores the least fixed point formulas that have been unfolded on the branch ending in v
- Γ is an SCloG sequent, which is sometimes called the sequent at v

If $l_V(v) \in L_{\mathcal{S}}$ then v is called a *sequent node*, and if $l_V(v) = \wedge$ then v is called a *conjunctive-node*.
The edge label set L_E consists of symbols representing clo applications in $\{\mathsf{clo}_\mathsf{x} \mid \mathsf{x} \in N\}$, other CloG rule names, and proof branches left or right. So the full set is defined as $L_E = \{\mathsf{clo}_\mathsf{x} \mid \mathsf{x} \in N\} \cup \{\vee, \wedge, ;, \sqcup, \sqcap, ?, !, *, \mathsf{left}, \mathsf{right}\}$

In order to succinctly refer to the formulas to which certain rules can be applied, some notation is defined. This helps define rules so that they cannot be applied unless rules of lower priority also cannot be applied.

Definition 5.6. The following definitions are for annotated formulas of a given form:

$\vee$-formula:	A formulas of the form $(\varphi \vee \psi)^\mathsf{a}$
$\wedge$-formula:	A formulas of the form $(\varphi \wedge \psi)^\mathsf{a}$
$\sqcup$-formula:	A formulas of the form $(\langle \gamma \sqcup \delta \rangle \varphi)^\mathsf{a}$
$\sqcap$-formula:	A formulas of the form $(\langle \gamma \sqcap \delta \rangle \varphi)^\mathsf{a}$
$;$-formula:	A formulas of the form $(\langle \gamma ; \delta \rangle \varphi)^\mathsf{a}$
$?$-formula:	A formulas of the form $(\langle \gamma ? \rangle \varphi)^\mathsf{a}$
$!$-formula:	A formulas of the form $(\langle \gamma ! \rangle \varphi)^\mathsf{a}$
$*$-formula:	A formulas of the form $(\langle \gamma^* \rangle \varphi)^\mathsf{a}$
$\times$-formula:	A formulas of the form $(\langle \gamma^\times \rangle \varphi)^\mathsf{a}$

$\vee, \sqcap, *$-formulas will be used to refer to formulas that are either a $\vee$-formula, $\sqcap$-formula, or $*$-formula

A definition is needed to stop further possible rule applications when the branch can be ended successfully by an axiom or discharged clo application. For this, a successful leaf node is defined. A node of this kind may contain a sequent to which other rules can be applied but, due to being this kind of node, the rules are no longer applicable.

Definition 5.7. In an SCloG search tree $T := (V, E, l_V, l_E)$, A vertex $v \in V$ is a *successful leaf node* if $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma)$ and $l_V(v)$ meets at least one of the two following conditions:

1. Γ contains a pair of annotated formulas of the form $(p)^{\mathsf{a}}$ and $(\neg p)^{\mathsf{b}}$
2. Γ contains a $\times$-formula, $\langle\gamma^{\times}\rangle\varphi^{\mathsf{axb}}$, such that $\mathfrak{h}(\mathsf{x}) = (\langle\gamma^{\times}\rangle\varphi^{\mathsf{a}}, \Gamma')$ and $\forall\psi^{\mathsf{c}} \in \Gamma'\ \exists\mathsf{d} \in N^* :\ \mathsf{c} \subseteq \mathsf{d}$ and $\psi^{\mathsf{d}} \in \Gamma$

A definition is given for matching applications of $\mathsf{clo}_{\mathsf{x}}$ which were described in Section 4.5. The definition for a cycling unfolding is only described in its effect, without a concrete definition, and therefore cannot yet be implemented. It is used here to show how it will factor into the algorithm if a correct definition is found. The ordering $\mathsf{x} \prec_{N_V} \mathsf{x}'$ is defined if x occurs before x' in the sequence N_V.

Definition 5.8. Let $T := (V, E, l_V, l_E)$ be an SCloG search tree. If there is a vertex $v \in V$ where $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma)$ and $\exists(\langle\gamma^{\times}\rangle\varphi)^{\mathsf{a}} \in \Gamma$, $\exists\mathsf{x}, \mathsf{x}' \in N_V :\ \mathsf{x}' \prec_{N_V} \mathsf{x}$ such that $\mathfrak{h}(\mathsf{x}) = ((\langle\gamma^{\times}\rangle\varphi)^{b}, \Phi')$, $\mathfrak{h}(\mathsf{x}') = ((\langle\gamma^{\times}\rangle\varphi)^{c}, \Phi'')$, and $\Phi_\Gamma = \{\psi \mid (\psi)^{\mathsf{d}} \in \Phi'\} = \{\psi \mid (\psi)^{\mathsf{d}} \in \Phi''\}$, then the names $\mathsf{x}, \mathsf{x}', \mathsf{x}''$ are matching unfoldings. The vertex v is a *cycling unfolding* if the annotations in Γ will not lead forms of derivation that cannot be found using only the annotations after the application of $\mathsf{clo}_{\mathsf{x}}$.

5.1.1 SCloG Search Rules

As stated earlier, the search tree is built from the root to the leaves by adding new vertices and edges dependent on the vertex labels. What follows in Definition 5.10 is a set of conditions on an SCloG search tree which define where vertices should have outgoing branches. These conditions are organised into what are referred to as *search rules*. If a search rule is *applicable* to a given sequent node, then the node has an outgoing edge with a label matching the name of that search rule. The label of the child node is then defined dependent on the form of the parent label. All rules also require the vertex to not be a successful leaf or cycling unfolding. Both of these conditions do not imply that no further rule applications can be applied. However, in the case of a successful leaf, no further search branches need be added as the algorithm should halt here and close the proof branch. In a similar vein, following a cycling unfolding, no new search branches will lead to further successful branches that have not already been found early on in the cycle. Therefore, in both cases the search branch should be concluded and these conditions need to be included in all rules.

Definition 5.9. The *active formulas* of a rule application are the formulas in the parent sequent node label of the rule that do not appear in the next successive sequent node label due to being replaced by new formulas by the rule.

Definition 5.10.
The first search rule, clo, requires the sequent at a vertex to contain a greatest fixed point formula. Additionally, all names in the annotation of the fixed point formula should precede the fixed point based on the partial ordering $\preceq$ from Definition 2.4. The new sequent contains the unfolded fixed point with a new name added to the annotation in the same way as in the CloG clo rule. The new name is added to the sequence of names on the branch N_V by sequence concatenation. The unfolding is recorded by adding the mapping from the new name to the unfolding in the partial injective function $\mathfrak{h}$. Another edge will be added for all greatest fixed point formulas in the sequent at the given sequent node.

Search rule clo: *active formula:* $\langle\gamma^{\times}\rangle\varphi^{\mathsf{a}}$

$\forall v \in V$

If: $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; \langle\gamma^{\times}\rangle\varphi^{\mathsf{a}}; \Gamma')$
and $\mathsf{a} \preceq \langle\gamma^{\times}\rangle\varphi$
and v is not a successful leaf node
and v is not a cycling unfolding

Then: $\exists v' \in V$ and $\exists\mathsf{x} \in N$
such that $(v, v') \in E$
and $\mathsf{x} \notin N_V$
and $l_V(v') = (N_V; \mathsf{x}, \mathfrak{h} \cup \{\mathsf{x} \mapsto (\langle\gamma^{\times}\rangle\varphi^{\mathsf{a}}, \Gamma; \langle\gamma^{\times}\rangle\varphi^{\mathsf{a}}; \Gamma')\}, \mathcal{H}, dup(\Gamma; \Gamma'; (\varphi \wedge \langle\gamma\rangle\langle\gamma^{\times}\rangle\varphi)^{\mathsf{ax}}))$
and $l_E((v, v')) =\ \mathsf{clo}_{\mathsf{x}}$

The $\vee$ search rule is a simple rule so can only be applied to the first possible $\vee$-formula in the sequent, this is imposed by the condition Γ contains no $\vee$-formula. The other rules have a similar condition for their own form of proposition.

Search rule $\vee$: *active formula:* $(\varphi \vee \psi)^{\mathsf{a}}$

$\forall v \in V$

If: $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\varphi \vee \psi)^{\mathsf{a}}; \Gamma')$

and Γ contains no $\vee$-formula

and v is not a successful leaf node

and v is not a cycling unfolding

Then: $\exists v' \in V$

such that $(v, v') \in E$

and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, \mathit{dup}(\Gamma; \Gamma'; (\varphi)^{\mathsf{a}}, (\psi)^{\mathsf{a}}))$

and $l_E((v, v')) = \vee$

The $\sqcup$ search rule is the first simple rule that has the condition that it cannot be applied if another previous rule is also applicable. This is imposed by the condition Γ, Γ' contain no $\vee$-formulas.

Search rule $\sqcup$: *active formula:* $(\langle \gamma \sqcup \delta \rangle \varphi)^{\mathsf{a}}$

$\forall v \in V$

If: $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\langle \gamma \sqcup \delta \rangle \varphi)^{\mathsf{a}}; \Gamma')$

and Γ contains no $\sqcup$-formula

and Γ, Γ' contain no $\vee$-formulas

and v is not a successful leaf node

and v is not a cycling unfolding

Then: $\exists v' \in V$

such that $(v, v') \in E$

and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, \mathit{dup}(\Gamma; \Gamma'; (\langle \gamma \rangle \varphi \vee \langle \delta \rangle \varphi)^{\mathsf{a}}))$

and $l_E((v, v')) = \sqcup$

Search rule ; : *active formula:* $(\langle \gamma; \delta \rangle \varphi)^{\mathsf{a}}$

$\forall v \in V$

If: $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\langle \gamma; \delta \rangle \varphi)^{\mathsf{a}}; \Gamma')$

and Γ contains no ;-formula

and Γ, Γ' contain no $\vee, \sqcup$-formulas

and v is not a successful leaf node

and v is not a cycling unfolding

Then: $\exists v' \in V$

such that $(v, v') \in E$

and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, \mathit{dup}(\Gamma; \Gamma'; (\langle \gamma \rangle \langle \delta \rangle \varphi)^{\mathsf{a}}))$

and $l_E((v, v')) = ;$

Search rule ?: *active formula:* $(\langle\psi?\rangle\varphi)^{\mathsf{a}}$

$\forall v \in V$

If: $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\langle\psi?\rangle\varphi)^{\mathsf{a}}; \Gamma')$
and Γ contains no ?-formula
and Γ, Γ' contain no $\vee, \sqcup,$;-formulas
and v is not a successful leaf node
and v is not a cycling unfolding

Then: $\exists v' \in V$
such that $(v, v') \in E$
and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, dup(\Gamma; \Gamma'; (\psi \wedge \varphi)^{\mathsf{a}}))$
and $l_E((v, v')) = ?$

Search rule !: *active formula:* $(\langle\psi!\rangle\varphi)^{\mathsf{a}}$

$\forall v \in V$

If: $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\langle\psi!\rangle\varphi)^{\mathsf{a}}; \Gamma')$
and Γ contains no !-formula
and Γ, Γ' contain no $\vee, \sqcup, ;, ?$-formulas
and v is not a successful leaf node
and v is not a cycling unfolding

Then: $\exists v' \in V$
such that $(v, v') \in E$
and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, dup(\Gamma; \Gamma'; (\psi \vee \varphi)^{\mathsf{a}}))$
and $l_E((v, v')) = !$

Search rule $\sqcap$: *active formula:* $(\langle\gamma \sqcap \delta\rangle\varphi)^{\mathsf{a}}$

$\forall v \in V$

If: $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\langle\gamma \sqcap \delta\rangle\varphi)^{\mathsf{a}}; \Gamma')$
and Γ contains no $\sqcap$-formula
and Γ, Γ' contain no $\vee, ;, ?, !$-formulas
and v is not a successful leaf node
and v is not a cycling unfolding

Then: $\exists v' \in V$
such that $(v, v') \in E$
and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, dup(\Gamma; \Gamma'; (\langle\gamma\rangle\varphi \wedge \langle\delta\rangle\varphi)^{\mathsf{a}}))$
and $l_E((v, v')) = \sqcap$

Search rule $*$: *active formula:* $(\langle\gamma^*\rangle\varphi)^{\mathsf{a}}$

$\forall v \in V$
If: $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\langle\gamma^*\rangle\varphi)^{\mathsf{a}}; \Gamma')$
and $\mathsf{a} \preceq \langle\gamma^*\rangle\varphi$
and Γ contains no $*$-formula
and Γ, Γ' contain no $\vee, \sqcup, ;, ?, !$-formulas
and v is not a successful leaf node
and v is not a cycling unfolding
Then: $\exists v' \in V$
such that $(v, v') \in E$
and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H} \cup \{((\langle\gamma^*\rangle\varphi)^{\mathsf{a}}, \Phi^N_{\Gamma;(\langle\gamma^*\rangle\varphi)^{\mathsf{a}};\Gamma'})\}, dup(\Gamma; \Gamma'; (\varphi \vee \langle\gamma\rangle\langle\gamma^*\rangle\varphi)^{\mathsf{a}}))$
and $l_E((v, v')) = *$

The $\wedge$ rule from CloG splits a CloG proof into two proof branches that must both lead to a sub-proof. The search rule $\wedge$ therefore also splits the search tree into two branches that represent the two CloG branches. The search rule also introduces a conjunctive node so that the search algorithm can be easily defined to require a successful search from both the left and right edges.

Search rule $\wedge$: *active formula:* $(\psi \wedge \varphi)^{\mathsf{a}}$

$\forall v \in V$
If: $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\psi \wedge \varphi)^{\mathsf{a}}; \Gamma')$
and Γ contains no $\wedge$-formula
and Γ, Γ' contain no $\vee, \sqcup, ;, ?, !$-formulas
and Γ, Γ' contain no $*$-formula or $\forall((\langle\gamma^*\rangle\varphi)^{\mathsf{a}}) \in \Gamma, \Gamma' : ((\langle\gamma^*\rangle\varphi)^{\mathsf{a}}, \Phi^N_{\Gamma;(\langle\gamma^*\rangle\varphi)^{\mathsf{a}};\Gamma'}) \in \mathcal{H}$
and v is not a successful leaf node
and v is not a cycling unfolding
Then: $\exists v', v^l, v^r \in V$
such that $(v, v'), (v', v^l), (v', v^r) \in E$
and $l_V(v') = \wedge$
and $l_V(v^l) = (N_V, \mathfrak{h}, \mathcal{H}, dup(\Gamma; \Gamma'; (\varphi)^{\mathsf{a}}))$
and $l_V(v^r) = (N_V, \mathfrak{h}, \mathcal{H}, dup(\Gamma; \Gamma'; (\psi)^{\mathsf{a}}))$
and $l_E((v, v')) = \wedge$
and $l_E((v', v^l)) = \mathsf{left}$
and $l_E((v', v^r)) = \mathsf{right}$

The mod_m search rule also includes applications of weak in order to reduce down the matching atomic modalities to which the CloG mod_m rule can be applied. The removal of the side formulas make the mod_m search rule another that must be attempted for all available pairs of atomic modalities.

Search rule mod_m: *active formulas:* $(\langle a\rangle\varphi)^{\mathsf{a}}; \Gamma', (\langle a^d\rangle\psi)^{\mathsf{b}}$

$\forall v \in V$

If: $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\langle a\rangle\varphi)^{\mathsf{a}}; \Gamma'; (\langle a^d\rangle\psi)^{\mathsf{b}}; \Gamma'')$ or $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\langle a^d\rangle\varphi)^{\mathsf{a}}; \Gamma'; (\langle a\rangle\psi)^{\mathsf{b}}; \Gamma'')$

and Γ, Γ' contain no $\vee, \sqcup, ;, ?, \sqcap, \wedge$-formulas

and Γ, Γ' contain no $*$-formula or $\forall((\langle\gamma^*\rangle\varphi)^{\mathsf{a}}) \in \Gamma, \Gamma' : ((\langle\gamma^*\rangle\varphi)^{\mathsf{a}}, \Phi^N_{\Gamma;(\langle\gamma^*\rangle\varphi)^{\mathsf{a}};\Gamma'}) \in \mathcal{H}$

and v is not a successful leaf node

and v is not a cycling unfolding

Then: $\exists v' \in V$

such that $(v, v') \in E$

and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, (\varphi)^{\mathsf{a}}; (\psi)^{\mathsf{b}})$

and $l_E((v, v')) = \mathsf{mod}_m$

A definition is then given for a well formed SCloG tree for a given root sequent Γ_0. The condition requires the sequent at the root vertex of the tree to be Γ_0 and then the rest of the tree is built up from there by the search rules.

Definition 5.11. Let $\Gamma_0 \in \mathcal{S}$ be an SCloG sequent. A *well formed SCloG search tree for* Γ_0 is an SCloG search tree $T := (V, E, l_V, l_E)$ that satisfies:

- $l_V(v_0) = (\{\}, \{\}, \{\}, \Gamma_0)$ where v_0 is the root of T.
- All other vertices and edges in the tree are added by search rules in Definition 5.10.

A definition can now be given for when an SCloG search tree is successful in finding a proof for the root sequent.

Definition 5.12. Let $T := (V, E, l_V, l_E)$ be a well formed SCloG search tree for Γ_0. A sequent node $v \in V$ is a *successful node* if it is either a successful leaf node or there is at least one other successful node $v' \in V$ such that $(v, v') \in E$. A conjunctive node $v_\wedge \in V$ is a successful node if for all $v' \in V$ where $(v, v') \in E$, v' is a successful node. If the root vertex of the tree is a successful node, then the search tree is successful.

An SCloG derivation would then represent one possible CloG proof from the successful SCloG search tree by containing only successful nodes, all sequent nodes only having one outgoing edge, and conjunctive nodes all having both outgoing edges. In the next section it is shown that given a successful SCloG proof search tree T for Γ_0, a CloG proof exists for the CloG sequent $\Phi^N_{\Gamma_0}$. This proves the soundness of SCloG relative to CloG.

5.2 Soundness for SCloG

If a well formed SCloG search tree $T := (V, E, l_V, l_E)$ is successful, then there exists a well formed proof in CloG for the sequent at the root of T. A transformation from an SCloG derivation in T to the equivalent proof in CloG is therefore shown in this section. The function executes the transformation starting from the root and moving up to the leaves of an SCloG derivation using recursion, before converting any applications of clo that are not discharged at a leaf node into applications of $\times$. The transformation is not deterministic as there may be more than one SCloG derivation in T and only one is needed to find a CloG proof.

The main transformation function is denoted *t2p* which takes as input a successful SCloG search tree $T := (V, E, l_V, l_E)$ with a root SCloG sequent Γ_0 and outputs a CloG proof $\mathcal{P}$ for the CloG sequent $\Phi^N_{\Gamma_0}$. The function is supported by the conversion function *n2p* which takes a successful node $v \in V$ where $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma)$ and outputs a partial CloG proof $\mathcal{P}_\Gamma$ for the CloG sequent Φ^N_Γ. This is a partial proof as it has the structure of a CloG proof but not all applications of clo may be discharged. The cleanup function denoted *remClo* takes a partial CloG proof $\mathcal{P}_\Gamma$ for the sequent Γ and outputs the proof $\mathcal{P}$ which is the same but with clo applications that are not discharged converted to $\times$. The application $t2p(T)$ is equivalent to $remClo(n2p(v))$ where v is the root of T.

For an application of $n2p(v)$, where $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma)$, if v is a successful leaf node do one of the following:

- If Γ contains a pair of annotated formulas of the form $(p)^{\mathsf{a}}$ and $(\neg p)^{\mathsf{b}}$. Then, apply CloG rules weak and exp above the CloG sequent $\Phi^N_\Gamma := \Phi \cup \{(p)^{\mathsf{a}}, (\neg p)^{\mathsf{b}}\}$ to reach the CloG sequent $\{(p)^{\epsilon}, (\neg p)^{\epsilon}\}$ and finally apply the CloG axiom $\mathsf{Ax1}$. This gives a proof segment of the following form (Double lines represent zero or more applications of a CloG rule):

$$\cfrac{\cfrac{\cfrac{}{(p)^{\epsilon}, (\neg p)^{\epsilon}}\,\mathsf{Ax1}}{(p)^{\mathsf{a}}, (\neg p)^{\mathsf{b}}}\,\mathsf{exp}}{\Phi, (p)^{\mathsf{a}}, (\neg p)^{\mathsf{b}}}\,\mathsf{weak}$$

- Otherwise, If Γ contains a $\times$-formula, $\langle\gamma^{\times}\rangle\varphi^{\mathsf{axb}}$, such that $\mathfrak{h}(\mathsf{x}) = (\langle\gamma^{\times}\rangle\varphi^{\mathsf{a}}, \Gamma')$ and $\forall\psi^{\mathsf{c}} \in \Gamma'\ \exists\mathsf{d} \in N^*\ \forall\mathsf{x} \in \mathsf{c} : \mathsf{x} \in \mathsf{d}$ and $\psi^{\mathsf{d}} \in \Gamma$. Then, apply CloG weak to the CloG sequent Φ^N_Γ to reduce it to only annotated formulas where there is the same formula in $\Phi^N_{\Gamma'}$ but with a possible subset annotation. Then apply CloG exp to remove all names not in annotations of $\Phi^N_{\Gamma'}$ aside from the additions name x on the fixed point formula $\langle\gamma^{\times}\rangle\varphi$. The CloG proof segment can then end with discharging $\mathsf{clo}_{\mathsf{x}}$. The proof branch in this case has the form:

$$\cfrac{\cfrac{[\Phi^N_{\Gamma'}\backslash\{(\langle\gamma^{\times}\rangle\varphi)^{\mathsf{a}}\}, (\langle\gamma^{\times}\rangle\varphi)^{\mathsf{ax}}]^{\mathsf{x}}}{\{(\varphi)^{\mathsf{a}} \mid (\varphi)^{\mathsf{a}} \in \Phi^N_\Gamma, \exists(\varphi)^{\mathsf{b}} \in \Phi^N_{\Gamma'}\}}\,\mathsf{exp}}{\Phi^N_\Gamma}\,\mathsf{weak}$$

For an application of $n2p(v)$, and given $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma)$, if v is not a successful leaf node but just a successful node, select one successful node $v' \in V$ where $(v, v') \in E$. Then do one of the following:

- If $l_E((v, v')) = \mathsf{mod}_m$, then $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, (\varphi)^{\mathsf{a}}; (\psi)^{\mathsf{b}})$ and either $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma'; (\langle a\rangle\varphi)^{\mathsf{a}}; \Gamma''; (\langle a^d\rangle\psi)^{\mathsf{b}}; \Gamma''')$ or $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma'; (\langle a^d\rangle\varphi)^{\mathsf{a}}; \Gamma''; (\langle a\rangle\psi)^{\mathsf{b}}; \Gamma''')$. Hence, Φ^N_Γ is the same in both cases of $l_V(v)$. Apply weak on Φ^N_Γ so that the sequent only contains $(\langle a\rangle\varphi)^{\mathsf{a}}$ and $(\langle a^d\rangle\psi)^{\mathsf{b}}$. After this, apply mod_m and put the proof $n2p(v')$ above. This gives a CloG proof segment of the form:

$$\cfrac{\cfrac{n2p(v')}{(\langle a\rangle\varphi)^{\mathsf{a}}, (\langle a^d\rangle\psi)^{\mathsf{b}}}\,\mathsf{mod}_m}{\Gamma', \Gamma'', \Gamma''', (\langle a\rangle\varphi)^{\mathsf{a}}, (\langle a^d\rangle\psi)^{\mathsf{b}}}\,\mathsf{weak}$$

- If $l_E((v, v')) \in \{\wedge\}$ then $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma)$ and there are nodes $v^l, v^r \in V$ with relations $(v', v^l), (v', v^r) \in E$. Apply the $l_E((v, v'))$ rule to Φ^N_Γ and put $n2p(v^l)$ on the left side of the inference then $n2p(v^r)$ on the right. This gives a proof segment of the form:

$$\cfrac{n2p(v^l) \qquad n2p(v^r)}{\Phi^N_\Gamma}\,l_E((v, v'))$$

- If $l_E((v, v')) \notin \{\mathsf{mod}_m, \wedge\}$ then apply the rule $l_E((v, v'))$ to Φ^N_Γ with the proof $n2p(v')$ above. This proof segment has the form:

$$\cfrac{n2p(v')}{\Phi^N_\Gamma}\,l_E((v, v'))$$

Lastly, there is the application of $remClo(\mathcal{P}_\Gamma)$. For all applications of clo_x in $\mathcal{P}_\Gamma$, if there is no branch that ends in $[\Phi]^\mathsf{x}$ for some sequent Φ, the application of clo_x should be replaced by $\times$. To do this, first change the application of clo_x to an application of $\times$. Then, remove all appearances of x in an annotations of the sub-proof rooted at the sequent where clo_x. Lastly, now that all the names are removed from applications, remove any applications of exp where the sequent above is equal to the one below, also only in the sub-proof, and leave only one of the sequents in its place.

5.3 Pseudo-Code Definition

The interpretation of the algorithm into pseudo-code requires both a representation of the SCloG search tree structure and of CloG. The main pseudo-code flow uses a recursive function to explore the theoretical SCloG search tree for a successful branch with a depth first search strategy. The cycling unfolding condition is not included in the pseudo-code as it does not yet have an implementable condition.

At each sequent node, due to the edge construction rules, there will only be outgoing edges for the $\mathsf{clo}_{\mathsf{x}}$ rule and one other rule. For the purpose of the search goal, the order in which the edges are selected does not matter, but an efficient strategy can be selected. In this case, the pseudo-code will always search along the $\mathsf{clo}_{\mathsf{x}}$ edges first, in the order that the active fixed point formulas appear in the sequent. For the other rules, there will only be one other branch to search, aside from for mod_m. For mod_m, the order is chosen based on the first applicable pair of modalities, non-dual first and then dual. So for example in the sequent $\langle a^d\rangle p^\epsilon; \langle a\rangle p^\epsilon; \langle b\rangle p^\epsilon; \langle a^d\rangle q^\epsilon; \langle b^d\rangle q^\epsilon$, the pair $\langle a\rangle p^\epsilon, \langle a^d\rangle p^\epsilon$ would be chosen first. This would be followed by $\langle a\rangle p^\epsilon, \langle a^d\rangle q^\epsilon$ and then $\langle b\rangle p^\epsilon, \langle b^d\rangle q^\epsilon$.

Most of the transformation from the SCloG tree to CloG is also performed as the recursive flow returns up a successful branch, rather than transforming after the fact. At the end of the search, the cleanup step to remove non-discharged $\mathsf{clo}_{\mathsf{x}}$ applications is executed to retrieve a well formed CloG proof.

5.3.1 Data Types

A CloG proof in this pseudo-code, referred to as a "CloGProof", is represented using a sequence of alternating sequents and rules that are joined by + symbols and ending in an axiom (leading to an empty sequent) or discharged closure (a sequent annotated by the closure name). The sequents (which do not have a named type for CloGProof's) are sets of annotated formulas (typed as "Formula"). The rules (typed as "Rule") are assumed to have been defined as constants aside from $\mathsf{clo}_{\mathsf{x}}$ which comes with a name (typed as "Name"). The braches of the CloG proof tree are composed together into a sequence starting with the branch from the root sequent and after any $\wedge$ rule, the left branch comes first and the right branch comes after that branch has ended in one or more other branches. This forms the full proof. The CloGProof type can also be the constant "noProof" to identify a failed proof search.

See the following example CloGProof:
$\{(\langle a^{\mathsf{x}}\rangle\neg p)^\epsilon, (\langle a^{d^*}\rangle p)^\epsilon\} + \mathsf{clo}_{\mathsf{x}} +$
$\{(\neg p \wedge \langle a\rangle\langle a^{\mathsf{x}}\rangle\neg p)^{\mathsf{x}}, (\langle a^{d^*}\rangle p)^\epsilon\} + * +$
$\{(\neg p \wedge \langle a\rangle\langle a^{\mathsf{x}}\rangle\neg p)^{\mathsf{x}}, (p \vee \langle a^d\rangle\langle a^{d^*}\rangle p)^\epsilon\} + \vee +$
$\{(\neg p \wedge \langle a\rangle\langle a^{\mathsf{x}}\rangle\neg p)^{\mathsf{x}}, p^\epsilon, (\langle a^d\rangle\langle a^{d^*}\rangle p)^\epsilon\} + \wedge +$
$\{(\neg p)^{\mathsf{x}}, p^\epsilon, (\langle a^d\rangle\langle a^{d^*}\rangle p)^\epsilon\} + \mathsf{weak} +$
$\{(\neg p)^{\mathsf{x}}, p^\epsilon\} + \mathsf{exp} +$
$\{(\neg p)^\epsilon, p^\epsilon\} + \mathsf{Ax1} +$
$\{\} +$
$\{(\langle a\rangle\langle a^{\mathsf{x}}\rangle\neg p)^{\mathsf{x}}, p^\epsilon, (\langle a^d\rangle\langle a^{d^*}\rangle p)^\epsilon\} + \mathsf{weak} +$
$\{(\langle a\rangle\langle a^{\mathsf{x}}\rangle\neg p)^{\mathsf{x}}, (\langle a^d\rangle\langle a^{d^*}\rangle p)^\epsilon\} + \mathsf{mod}_m +$
$[\{(\langle a^{\mathsf{x}}\rangle\neg p)^{\mathsf{x}}, (\langle a^{d^*}\rangle p)^\epsilon\}]^{\mathsf{x}}$

Which represents the CloG proof tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(\neg p)^\epsilon, p^\epsilon}\,\mathsf{Ax1}}{(\neg p)^{\mathsf{x}}, p^\epsilon}\,\mathsf{exp}}{(\neg p)^{\mathsf{x}}, p^\epsilon, (\langle a^d\rangle\langle a^{d^*}\rangle p)^\epsilon}\,\mathsf{weak} \qquad \dfrac{\dfrac{[(\langle a^{\mathsf{x}}\rangle\neg p)^{\mathsf{x}}, (\langle a^{d^*}\rangle p)^\epsilon]^{\mathsf{x}}}{(\langle a\rangle\langle a^{\mathsf{x}}\rangle\neg p)^{\mathsf{x}}, (\langle a^d\rangle\langle a^{d^*}\rangle p)^\epsilon}\,\mathsf{mod}_m}{(\langle a\rangle\langle a^{\mathsf{x}}\rangle\neg p)^{\mathsf{x}}, p^\epsilon, (\langle a^d\rangle\langle a^{d^*}\rangle p)^\epsilon}\,\mathsf{weak}}{(\neg p \wedge \langle a\rangle\langle a^{\mathsf{x}}\rangle\neg p)^{\mathsf{x}}, p^\epsilon, (\langle a^d\rangle\langle a^{d^*}\rangle p)^\epsilon}\,\wedge}{(\neg p \wedge \langle a\rangle\langle a^{\mathsf{x}}\rangle\neg p)^{\mathsf{x}}, (p \vee \langle a^d\rangle\langle a^{d^*}\rangle p)^\epsilon}\,\vee}{(\neg p \wedge \langle a\rangle\langle a^{\mathsf{x}}\rangle\neg p)^{\mathsf{x}}, (\langle a^{d^*}\rangle p)^\epsilon}\,*}{(\langle a^{\mathsf{x}}\rangle\neg p)^\epsilon, (\langle a^{d^*}\rangle p)^\epsilon}\,\mathsf{clo}_{\mathsf{x}}$$

To represent the SCloG search tree, the structure is maintained by the recursive tree formed when executing the main function. The elements of a sequent node are the input to the main function. The sequent Γ is a list of annotated formulas (typed "Sequent"). To represent the fixed point unfoldings found in $\mathcal{H}$ and $\mathfrak{h}$, "FpSeq" is used. This type is a tuple containing an annotated fixed point formula that was unfolded, and a set of annotated formulas matching those in the sequent in which the formula was unfolded. Then $\mathcal{H}$ is a set of "FpSeq"s and $\mathfrak{h}$ is a map from "Name"s to "FpSeq"s.

The pseudo-code also utilises the value "null" to represent no value for a variable or function return.

5.3.2 Functions

The proof search uses several auxiliary functions for finding accessible edges and transforming to the CloG proof. These use list and set comprehension along with pattern matching. In these, the following symbols are used for arbitrary values:

- φ, ψ for propositions
- γ, δ for games
- p for an atomic proposition
- a for an atomic game
- x for an individual clo name
- $\mathsf{a}, \mathsf{b}, \mathsf{c}$ for formula annotations
- _ for any arbitrary value

The partial ordering on fixed point formulas $\preceq$, from Definition 2.4, is also used. The pseudo-code for the auxiliary function canDischargeClosure can be found in Algorithm 1, and canApply can be found in Algorithm 2.

Algorithm 1: canDischargeClosure(): Find the name, if any, of a clo application that can be discharged at a given sequent.

input :

- sequent: The representation of Γ. A non-repeating list of CloG annotated formulas
- gfpUnfoldings: The representation of $\mathfrak{h}$. A map from each name in N_V to the "FpSeq" matching the greatest fixed point unfolding where the name was introduced

output: A name for a clo application that can be discharged due to a repeated sequent or null

```
begin
    repeats ← []
    for $(\langle\gamma^{\mathsf{x}}\rangle\varphi)^{\mathsf{a}}$ in sequent do
        for x in a do
            $(\psi)^{\mathsf{b}} \leftarrow$ gfpUnfoldings[x][0]
            if $\langle\gamma^{\mathsf{x}}\rangle\varphi = \psi$ and $\mathsf{bx} \subseteq \mathsf{a}$ and $\forall(\chi)^{\mathsf{c}} \in$ sequent $\exists(\chi)^{\mathsf{d}} \in$ gfpUnfoldings[x][1] : $\mathsf{c} \subseteq \mathsf{d}$ then
                return x
            end
        end
    end
    return null
end
```

The canApply function does not include the conditions for rule ordering or successful leaf conditions, as these are implicit from the main proof search function. The conditions are therefore closer to CloG rules, with the addition of the sequent repetition detection for $*$ and combination of structural rules into Ax1 and mod_m.

Algorithm 2: canApply(): Find if a rule can be applied to a sequent and to which formulas.

```
input :
    • rule: A CloG rule
    • sequent: The representation of $\Gamma$. A non-repeating sequence of CloG annotated formulas
    • gfpUnfoldings: The representation of $\mathfrak{h}$. A map from each name in $N_V$ to the “FpSeq” matching the
      greatest fixed point unfolding where the name was introduced
    • lfpUnfoldings: The representation of $\mathcal{H}$. A set of “FpSeq”s matching all of the least fixed point
      unfoldings that have happened in the branch
output: The list of formulas or pairs of formulas from the sequent that can be used as the active
        formulas for the given rule.
Begin
    switch rule do
        case Ax1 do
            for $\varphi^{\mathsf{a}}$ in sequent do
                if $\varphi := p$ then
                    for $\psi^{\mathsf{b}}$ in sequent do
                        if $\psi = \neg p$ then
                            return $[\langle p^{\mathsf{a}},\ (\neg p)^{\mathsf{b}}\rangle]$
                        end
                    end
                end
            end
            return []
        end
        case $\mathsf{clo_x}$ do
            return $\leftarrow [\varphi^{\mathsf{a}} \mid \varphi^{\mathsf{a}} \in \mathsf{sequent}\ \&\ \varphi := \langle\gamma^{\times}\rangle\psi\ \&\ \forall \mathsf{x} \in \mathsf{a} : \mathsf{gfpUnfoldings}[\mathsf{x}][0] \preceq \langle\gamma^{\times}\rangle\psi]$
        end
        case $*$ do
            for $\varphi^{\mathsf{a}}$ in sequent do
                if $\varphi := \langle\gamma^{*}\rangle\psi$ and $\forall \mathsf{x} \in \mathsf{a} : \mathsf{gfpUnfoldings}[\mathsf{x}][0] \preceq \langle\gamma^{*}\rangle\psi$ and $\langle\varphi^{\mathsf{a}}, \mathsf{sequent}\rangle \notin \mathsf{lfpUnfoldings}$
                 then
                    return $[\langle\gamma^{*}\rangle\psi^{\mathsf{a}}]$
                end
            end
            return []
        end
        case $\vee$ do
            for $\varphi^{\mathsf{a}}$ in sequent do
                if $\varphi := \psi \vee \chi$ then
                    return $(\psi \vee \chi)^{\mathsf{a}}$
                end
            end
            return []
        end
    end
```

Algorithm 2: canApply(): Find if a rule can be applied to a sequent and to which formulas. (Continued)

```
switch rule do
    case $\wedge$ do
        for $\varphi^{\mathsf{a}}$ in sequent do
            if $\varphi := \psi \wedge \chi$ then
                return $(\psi \wedge \chi)^{\mathsf{a}}$
            end
        end
        return []
    end
    case $\sqcap$ do
        for $\varphi^{\mathsf{a}}$ in sequent do
            if $\varphi := \langle \gamma \sqcap \delta \rangle \psi$ then
                return $(\langle \gamma \sqcap \delta \rangle \psi)^{\mathsf{a}}$
            end
        end
        return []
    end
    case $\sqcup$ do
        for $\varphi^{\mathsf{a}}$ in sequent do
            if $\varphi := \langle \gamma \sqcup \delta \rangle \psi$ then
                return $(\langle \gamma \sqcup \delta \rangle \psi)^{\mathsf{a}}$
            end
        end
        return []
    end
    case ; do
        for $\varphi^{\mathsf{a}}$ in sequent do
            if $\varphi := \langle \gamma ; \delta \rangle \psi$ then
                return $(\langle \gamma ; \delta \rangle \psi)^{\mathsf{a}}$
            end
        end
        return []
    end
    case ? do
        for $\varphi^{\mathsf{a}}$ in sequent do
            if $\varphi := \langle \chi ? \rangle \psi$ then
                return $(\langle \chi ? \rangle \psi)^{\mathsf{a}}$
            end
        end
        return []
    end
    case ! do
        for $\varphi^{\mathsf{a}}$ in sequent do
            if $\varphi := \langle \chi ! \rangle \psi$ then
                return $(\langle \chi ! \rangle \psi)^{\mathsf{a}}$
            end
        end
        return []
    end
end
```

Algorithm 2: canApply(): Find if a rule can be applied to a sequent and to which formulas. (Continued)

```
    switch rule do
        case $\mathsf{mod}_m$ do
            return $[\langle\varphi^{\mathsf{a}}, \psi^{\mathsf{b}}\rangle \mid \varphi^{\mathsf{a}}, \psi^{\mathsf{b}} \in \mathsf{sequent}\ \&\ \exists a \in \mathsf{G}_0 : (\varphi := \langle a\rangle\varphi'\ \&\ \psi := \langle a^d\rangle\psi')]$
        end
    end
end
```

5.3.3 Simple Function Definitions

The other functions do not require pseudo-code definitions as the input and output can be easily described. These functions are as follows:

- CloGProof `toAxiom`(Sequent seq): given a Sequent containing an annotated atomic proposition and its negation, returns the CloG proof branch with applications of weak and exp to remove other formulas and names ending in an application of Ax1. Example application:

$$\begin{aligned}\texttt{toAxiom}([(\neg q)^{x_0,x_1}, (\langle a^*\rangle p)^{\epsilon}, (q)^{\epsilon}]) &= \{(\neg q)^{x_0,x_1}, (\langle a^*\rangle p)^{\epsilon}, (q)^{\epsilon}\} \\ &\quad + \mathsf{weak} + \{(\neg q)^{x_0,x_1}, (q)^{\epsilon}\} \\ &\quad + \mathsf{exp} + \{(\neg q)^{x_0}, (q)^{\epsilon}\} \\ &\quad + \mathsf{exp} + \{(\neg q)^{\epsilon}, (q)^{\epsilon}\} \\ &\quad + \mathsf{Ax1} + \{\}\end{aligned}$$

- CloGProof `discahrgeClosure`(Sequent seq, Name name, FpSeq gfpUnfolding): Input a Sequent seq where the set of formulas in seq is a superset of the sequent in the greatest fixed point unfolding gfpUnfolding and also contains the fixed point in gfpUnfolding. Returns the CloG proof with applications of weak and exp to remove formulas not in gfpUnfolding and remove the names from the fixed point annotation in seq that are not the gfpUnfolding fixed point annotation. The branch then ends in the discharged closure labelled by name. Example application:

$$\begin{aligned} &\texttt{dischargeClosure}([(q)^{\mathsf{x}_2}, (\langle a^{\times}\rangle p)^{\mathsf{x}_0,\mathsf{x}_1,x_3}, (\langle r!\rangle q)^{\epsilon}, (\langle a^{d^*}\rangle\neg p)^{\epsilon}\},\ \mathsf{x}_1,\ \langle((\langle a^{\times}\rangle p)^{\mathsf{x}_0}, \{(\langle a^{\times}\rangle p)^{\mathsf{x}_0}, (\langle a^{d^*}\rangle\neg p)^{\epsilon}\rangle]) \\ = \; &\{((q)^{\mathsf{x}_2}, \langle a^{\times}\rangle p)^{\mathsf{x}_0,\mathsf{x}_1,x_3}, (\langle r!\rangle q)^{\epsilon}, (\langle a^{d^*}\rangle\neg p)^{\epsilon}\} \\ &+ \mathsf{exp} + \{((q)^{\mathsf{x}_2}, \langle a^{\times}\rangle p)^{\mathsf{x}_0,\mathsf{x}_1}, (\langle r!\rangle q)^{\epsilon}, (\langle a^{d^*}\rangle\neg p)^{\epsilon}\} \\ &+ \mathsf{weak} + \{(\langle a^{\times}\rangle p)^{\mathsf{x}_0,\mathsf{x}_1}, (\langle r!\rangle q)^{\epsilon}, (\langle a^{d^*}\rangle\neg p)^{\epsilon}\} \\ &+ \mathsf{weak} + [\{(\langle a^{\times}\rangle p)^{\mathsf{x}_0,\mathsf{x}_1}, (\langle a^{d^*}\rangle\neg p)^{\epsilon}\}]^{\mathsf{x}_1}\end{aligned}$$

- Sequent `apply`(Rule rule, Sequent seq, Formula|Tuple⟨Formula,Formula⟩ active): Input a Sequent seq containing the formula or pair of formulas in active, to which rule can be applied above. Returns the sequent list that will come above rule if it is applied to the formula or formulas in active as defined in Section 2. Example application:

$$\begin{aligned} &\texttt{apply}(?, [((\langle(q \wedge \neg r)!\rangle p)^{\epsilon}, (\langle(\neg q \vee r)?\rangle\neg p)^{\epsilon}],\ (\langle(\neg q \vee r)?\rangle\neg p)^{\epsilon}) \\ = \; &((\langle(q \wedge \neg r)!\rangle p)^{\epsilon}; ((\neg q \vee r) \wedge \neg p)^{\epsilon}\end{aligned}$$

- Tuple⟨Sequent,Sequent⟩ `applyConjunct`(Sequent seq, Formula active): The same as apply but only for the ∧ rule which returns the two upper sequents from the rule application in a tuple. Example application:

$$\begin{aligned} &\texttt{applyConjunct}([((\langle(q \wedge \neg r)!\rangle p)^{\epsilon}, ((\neg q \vee r) \wedge \neg p)^{\epsilon}],\ \{((\neg q \vee r) \wedge \neg p)^{\epsilon}\}) \\ = \; &\langle(((\langle(q \wedge \neg r)!\rangle p)^{\epsilon}; ((\neg q \vee r))^{\epsilon}\}, (((\langle(q \wedge \neg r)!\rangle p)^{\epsilon}; (\neg p)^{\epsilon}\}\rangle\end{aligned}$$

- CloGProof `weakenTo`(Sequent from, CloGProof to): Given a Sequent from where the set of formulas in the sequent is a superset of the set at the root of a CloG proof to, returns a CloG proof with applications of weak to get from the Sequent from below to the proof segment to. Example application:

$$\begin{aligned}
\texttt{weakenTo}([(q)^{\mathsf{x}_2}, (\neg q)^{x_0,x_1}, (\langle a^*\rangle p)^{\epsilon}, (q)^{\epsilon}], \{(\neg q)^{x_0,x_1}, (q)^{\epsilon}\} \quad + \quad & \mathsf{exp} \ + \ \{(\neg q)^{x_0}, (q)^{\epsilon}\} \\
& + \mathsf{exp} \ + \ \{(\neg q)^{\epsilon}, (q)^{\epsilon}\} \\
& + \mathsf{Ax1} \ + \ \{\}) \\
= \quad & \{(q)^{\mathsf{x}_2}, \{(\neg q)^{x_0,x_1}, (\langle a^*\rangle p)^{\epsilon}, (q)^{\epsilon}\} \\
& + \mathsf{weak} \ + \ \{\{(\neg q)^{x_0,x_1}, (\langle a^*\rangle p)^{\epsilon}, (q)^{\epsilon}\} \\
& + \mathsf{weak} + \{(\neg q)^{x_0,x_1}, (q)^{\epsilon}\} \\
& + \mathsf{exp} \ + \ \{(\neg q)^{x_0}, (q)^{\epsilon}\} \\
& + \mathsf{exp} \ + \ \{(\neg q)^{\epsilon}, (q)^{\epsilon}\} \\
& + \mathsf{Ax1} \ + \ \{\}
\end{aligned}$$

- CloGProof `removeNotDischargedClo`(CloGProof $\mathsf{partial}$): Given a partial CloG proof $\mathsf{partial}$ where some applications of $\mathsf{clo_x}$ may not be discharged. Returns the same proof but with those applications of $\mathsf{clo_x}$ without a discharge converted into applications of ×, the name x removed, and applications of exp that remove x also removed. Example application:

$$\begin{aligned}
\texttt{removeNotDischargedClo}(\quad & \{(\neg p)^{\epsilon}, (\langle (q \vee \neg q)!^{\times}\rangle p)^{\epsilon}\} \\
& + \mathsf{clo_x} \ + \ \{(\neg p)^{\epsilon}, (p \wedge \langle (q \vee \neg q)!\rangle\langle (q \vee \neg q)!^{\times}\rangle p)^{\mathsf{x}}\} \\
& + \wedge \\
& + \{(\neg p)^{\epsilon}, p^{\mathsf{x}}\} \\
& + \mathsf{exp} \ + \{(\neg p)^{\epsilon}, p^{\epsilon}\} \\
& + \mathsf{Ax1} \ + \ \{\} \\
& + \{(\neg p)^{\epsilon}, (\langle (q \vee \neg q)!\rangle\langle (q \vee \neg q)!^{\times}\rangle p)^{\mathsf{x}}\} \\
& + \mathsf{weak} \ + \ \{(\langle (q \vee \neg q)!\rangle\langle (q \vee \neg q)!^{\times}\rangle p)^{\mathsf{x}}\} \\
& + ! \ + \ \{((q \vee \neg q) \vee \langle (q \vee \neg q)!^{\times}\rangle p)^{\mathsf{x}}\} \\
& + \vee + \{((q \vee \neg q))^{\mathsf{x}}, (\langle (q \vee \neg q)!^{\times}\rangle p)^{\mathsf{x}}\} \\
& + \vee + \{(q^{\mathsf{x}}, (\neg q)^{\mathsf{x}}, (\langle (q \vee \neg q)!^{\times}\rangle p)^{\mathsf{x}}\} \\
& + \mathsf{weak} \ + \ \{q^{\mathsf{x}}, (\neg q)^{\mathsf{x}}\} \\
& + \mathsf{exp} \ + \ \{q^{\epsilon}, (\neg q)^{\mathsf{x}}\} \\
& + \mathsf{exp} \ + \ \{q^{\epsilon}, (\neg q)^{\epsilon}\} \\
& + \mathsf{Ax1} \ + \ \{\}) \\
= \quad & \{(\neg p)^{\epsilon}, (\langle (q \vee \neg q)!^{\times}\rangle p)^{\epsilon}\} \\
& + \ \times \ + \{(\neg p)^{\epsilon}, (p \wedge \langle (q \vee \neg q)!\rangle\langle (q \vee \neg q)!^{\times}\rangle p)^{\epsilon}\} \\
& + \wedge \\
& + \{(\neg p)^{\epsilon}, p^{\epsilon}\} \\
& + \mathsf{Ax1} \ + \ \{\} \\
& + \{(\neg p)^{\epsilon}, (\langle (q \vee \neg q)!\rangle\langle (q \vee \neg q)!^{\times}\rangle p)^{\epsilon}\} \\
& + \mathsf{weak} \ + \ \{(\langle (q \vee \neg q)!\rangle\langle (q \vee \neg q)!^{\times}\rangle p)^{\epsilon}\} \\
& + ! \ + \ \{((q \vee \neg q) \vee \langle (q \vee \neg q)!^{\times}\rangle p)^{\epsilon}\} \\
& + \vee + \{((q \vee \neg q))^{\epsilon}, (\langle (q \vee \neg q)!^{\times}\rangle p)^{\epsilon}\} \\
& + \vee + \{(q^{\epsilon}, (\neg q)^{\epsilon}, (\langle (q \vee \neg q)!^{\times}\rangle p)^{\epsilon}\} \\
& + \mathsf{weak} \ + \ \{q^{\epsilon}, (\neg q)^{\epsilon}\} \\
& + \mathsf{Ax1} \ + \ \{\}
\end{aligned}$$

5.3.4 Main Algorithm

The recursive proof search algorithm is found in the proofSearch() function of Algorithm 3 which cycles through the different possible rules, checking if they can be applied to any formulas and subsequently applying them if they can be. After each application the function is recursively called on the next sequent or sequents above and if no proof is returned, either the next possible rule is attempted or no proof is returned. If a proof is found, a segment of the CloG proof with the current sequent and rule is added to the proof and returned. The function will return a partial proof as some applications of $\mathsf{clo}_{\mathsf{x}}$ may not be discharged.

```
Algorithm 3: proofSearch(): Recursive proof search algorithm
  input :
    • sequent: The representation of $\Gamma$. A non-repeating sequence of CloG annotated formulas
    • branchNames: The representation of $N_V$. A list of the CloG names on the current search branch
    • gfpUnfoldings: The representation of $\mathfrak{h}$. A map from each name in $N_V$ to the "FpSeq" matching the
      greatest fixed point unfolding where the name was introduced
    • lfpUnfoldings: The representation of $\mathcal{H}$. A set of "FpSeq"s matching all of the least fixed point
      unfoldings that have happened in the branch
  output: Either a CloG proof (partial) or an indication of a failed proof search 'noProof'
  Begin
    List active ← canApply(Ax1, sequent, gfpUnfoldings, lfpUnfoldings) // First find if a
     branch can end in an axiom
    if active ! = [] then
      return sequent + toAxiom(sequent, active[0])
    end
    Name|List|null name ← canDischargeClosure(sequent, gfpUnfoldings) // Next find if any
     applications of $\mathsf{clo}_{\mathsf{x}}$ can be discharged on this branch
    if name is List then
      for fixedPointFormula in name do
        if isCyclingUnfolding(sequent, fixedPointFormula, branchNames, gfpUnfoldings) then
          return noProof ;     // Check for the possible cycling unfoldings found in
           canDischargeClosure
        end
      end
    else if name ! = null then
      return dischargeClosure(sequent, name, gfpUnfoldings[name])
    end
    List active ← canApply($\mathsf{clo}_{\mathsf{x}}$, sequent, gfpUnfoldings) // Find greatest fixed points which
     can be unfolded
    for fixedPointFormula in active do
      Name x ← getNewName(branchNames)
      Sequent nextSequent ← apply($\mathsf{clo}_{\mathsf{x}}$, sequent, fixedPointFormula)
      List newBranchNames ← branchNames + x
      Map[Name⇒FpSeq] newGfpUnfoldings ← gfpUnfoldings + {x ⇒ ⟨fixedPointFormula, sequent⟩}
      CloGProof possibleProof ← proofSearch(nextSequent, newBranchNames, newGfpUnfoldings,
       lfpUnfoldings)
      if possibleProof ! = noProof then
        return sequent + $\mathsf{clo}_{\mathsf{x}}$ + possibleProof
      end
    end
```

Algorithm 3: proofSearch(): Recursive proof search algorithm (Continued)

```
    for rule in [$\vee, \sqcup, ;, ?, !, \sqcap, *$] do
        List active ← canApply(rule, sequent, gfpUnfoldings, lfpUnfoldings) // Look to apply a
         set of simple rules
        if active  ! = [] then
            Sequent nextSequent ← apply(rule, sequent, active[0]})
            Set newLfpUnfoldings ← lfpUnfoldings
            if rule =  $*$ then
                newLfpUnfoldings ← newLfpUnfoldings + ⟨active[0], sequent⟩
            end
            CloGProof possibleProof ← proofSearch(nextSequent, branchNames, gfpUnfoldings,
             newLfpUnfoldings)
            if possibleProof ! = noProof then
                return sequent + rule + possibleProof
            return noProof
        end
    end
    List active ← canApply(rule, sequent, gfpUnfoldings, lfpUnfoldings)
    if active ! = [] then
        List conjunctSequent ← sequent
        Formula conjunction ← active[0]
        Tuple⟨Sequent,Sequent⟩ ⟨leftSequent, rightSequent⟩ ← applyConjunct(conjunctSequent,
         conjunction)
        CloGProof possibleLeftProof ← proofSearch(leftSequent, branchNames, gfpUnfoldings,
         lfpUnfoldings)
        if possibleLeftProof ! = noProof then
            CloGProof possibleRightProof ← proofSearch(rightSequent, branchNames, gfpUnfoldings,
             lfpUnfoldings)
            if possibleRightProof ! = noProof then
                return conjunctSequent + $\wedge$ + possibleLeftProof + possibleRightProof
            end
        end
        return noProof
    end
```

Algorithm 3: proofSearch(): Recursive proof search algorithm (Continued)

```
    for modalAtomPairs in canApply(mod$_m$, sequent, gfpUnfoldings, lfpUnfoldings) do
        Sequent nextSequent ← apply(mod$_m$, sequent, modalAtomPairs) // Lastly try
         applications of mod$_m$ on pairs of atomic modal formulas
        CloGProof possibleProof ← proofSearch(nextSequent, branchNames, gfpUnfoldings,
         lfpUnfoldings)
        if possibleProof ! = noProof then
            return sequent + weakenTo(sequent, {modalAtomPairs[0],modalAtomPairs[1]} + mod$_m$ +
             possibleProof)
        end
    end
    return noProof
end
```

The main function findProof() takes a root sequent with no annotations as input and then calls the proofSearch() function with that sequent and the other parameters empty as there is no branch history in the beginning. If a proof is found, the output of proofSearch() is then put through the removeNotDischargedClo() and returns the resulting well formed CloG proof. If no proof is found then the function outputs noProof.

Algorithm 4: proofSearch(): Main proof search algorithm (Continued)

```
input  : Sequent root: The root sequent for the proof to be found. Must have all empty annotations
output:
begin
    CloGProof incomplete ← proofSearch(root, [], {}, {})
    if incomplete = noProof then
        return noProof
    end
    return removeNotDischargedClo(incomplete)
end
```

6 Towards a Proof of Algorithm Termination

The algorithm uses a depth first search strategy where the search space is a well formed SCloG search tree. In order to show that the search algorithm terminates for any root sequent $\Gamma_0 \in \mathcal{S}$, it is sufficient to show that the search space is finite. This means showing that a well formed SCloG search tree for any root sequent Γ_0, has no infinitely extending branches and no vertex has infinitely many outgoing edges. The tree is therefore always finite in size. Throughout this section, the size or length of a set or sequent S is referred to with the notation $|S|$.

6.1 Complexity of a Formula

A function $\mathsf{c} : \mathcal{L}_{\mathsf{NF}} \mapsto \mathbb{N}$ is first defined to give the complexity of Game Logic propositions. This is inductively defined on the construction of a Game Logic proposition in $\mathcal{L}_{\mathsf{NF}}$ as follows:

Definition 6.1. With $p \in \mathsf{P}_0$, $a \in \mathsf{G}_0$, $\varphi, \psi \in \mathcal{L}_{\mathsf{NF}}$, and $\gamma, \delta \in \mathcal{G}_{\mathsf{NF}}$

$$\begin{array}{rcl}
\mathsf{c}(p) & = & 1 \\
\mathsf{c}(\neg p) & = & 1 \\
\mathsf{c}(\varphi \vee \psi) & = & \mathsf{c}(\varphi) + \mathsf{c}(\psi) + 1 \\
\mathsf{c}(\varphi \wedge \psi) & = & \mathsf{c}(\varphi) + \mathsf{c}(\psi) + 1 \\
\mathsf{c}(\langle a \rangle \varphi) & = & \mathsf{c}(\varphi) + 1 \\
\mathsf{c}(\langle a^d \rangle \varphi) & = & \mathsf{c}(\gamma) + 1 \\
\mathsf{c}(\langle \gamma ; \delta \rangle \varphi) & = & \mathsf{c}(\langle \gamma \rangle \langle \varphi \rangle \varphi) + 1 \\
\mathsf{c}(\langle \gamma \sqcup \delta \rangle \varphi) & = & \mathsf{c}(\langle \gamma \rangle \varphi \vee \langle \delta \rangle \varphi) + 1 \\
\mathsf{c}(\langle \gamma \sqcap \delta \rangle \varphi) & = & \mathsf{c}(\langle \gamma \rangle \varphi \wedge \langle \delta \rangle \varphi) + 1 \\
\mathsf{c}(\langle \gamma^* \rangle \varphi) & = & \mathsf{c}(\langle \gamma \rangle \mathsf{c}(\varphi)) \\
\mathsf{c}(\langle \gamma^\times \rangle \varphi) & = & \mathsf{c}(\langle \gamma \rangle \mathsf{c}(\varphi)) \\
\mathsf{c}(\langle \psi ? \rangle \varphi) & = & \mathsf{c}(\psi \wedge \varphi) + 1 \\
\mathsf{c}(\langle \psi ! \rangle \varphi) & = & \mathsf{c}(\psi \vee \varphi) + 1
\end{array}$$

This is defined in order to show that all SCloG search rules besides clo and $*$ will consistently reduce formulas down into formulas of smaller complexity. The definition of c is extended to SCloG-sequents.

Definition 6.2. Let $T := (V, E, l_V, l_E)$ by an SCloG search tree and let $v \in V$ be a vertex in the tree such that $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma)$. The complexity of Γ is defined as follows:

$$\mathsf{c}(\Gamma) = \sum_{(\varphi)^{\mathsf{a}} \ \in \ \Gamma} \mathsf{c}(\varphi)$$

Definition 6.3. Any SCloG rule in the set $\{\vee, \sqcap, ;, ?, !, \sqcap, \wedge, \mathsf{mod}_m\}$ is referred to as a *reduction rule.*

It now must be proven that each successor sequent node added by a reduction rule is of smaller complexity than the sequent at the node where the reduction rule was applied.

Lemma 6.1. Let $T = (V, E, l_V, l_E)$ be a well formed SCloG search tree for Γ_0. Let be a pair $v, v' \in V$ where $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma)$, $l_V(v') = (N'_V, \mathfrak{h}', \mathcal{H}', \Gamma')$ such that either $(v, v') \in E$ and $l_E((v, v')) \notin \{\mathsf{left}, \mathsf{right}, \mathsf{clo}, *, \wedge\}$, or $\exists v'' \in V : (v, v''), (v'', v') \in E$, $l_E((v, v'')) = \wedge$ and $l_E((v'', v')) \in \{\mathsf{left}, \mathsf{right}\}$. Then it is the case that $\mathsf{c}(\Gamma) < \mathsf{c}(\Gamma')$.

Proof. The property is shown case by case. All cases for complexity of formulas in Definition 6.1 are either 1 or other the sum of other complexities plus a natural number. Therefore, the property $\mathsf{c}(\varphi) > 0$ is used for any $\varphi \in \mathcal{L}_{\mathsf{NF}}$. It is also the case that $dup(\Gamma)$ can only remove formulas from Γ so the property $\mathsf{c}(dup(\Gamma)) \leq \mathsf{c}(\Gamma)$

Cases where $(v, v') \in E$ and $l_E((v, v')) \in \{\vee, \sqcap, ;, ?, !, \mathsf{mod}_m\}$:

- Case $l_E((v,v')) = \vee$
  $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\varphi \vee \psi)^{\mathsf{b}}; \Gamma')$ and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, \mathit{dup}(\Gamma; \Gamma'; (\varphi)^{\mathsf{b}}; (\psi)^{\mathsf{b}}))$

$$\begin{array}{rcl}
\mathsf{c}(\Gamma; (\varphi \vee \psi)^{\mathsf{b}}; \Gamma') & = & \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\varphi \vee \psi) \quad \text{(By Def. 6.2)} \\
& = & \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\varphi) + \mathsf{c}(\psi) + 1 \quad \text{(By Def. 6.1)} \\
& > & \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\varphi) + \mathsf{c}(\psi) \\
& = & \mathsf{c}(\Gamma; \Gamma'; (\varphi)^{\mathsf{b}}; (\psi)^{\mathsf{b}}) \quad \text{(By Def. 6.2)} \\
& \geq & \mathsf{c}(\mathit{dup}(\Gamma; \Gamma'; (\varphi)^{\mathsf{b}}; (\psi)^{\mathsf{b}})) \quad \text{(Given } \mathsf{c}(\mathit{dup}(\Gamma)) \leq \mathsf{c}(\Gamma)\text{)} \\
\\
\Rightarrow \mathsf{c}(\Gamma; (\varphi \vee \psi)^{\mathsf{b}}; \Gamma') & > & \mathsf{c}(\mathit{dup}(\Gamma; \Gamma'; (\varphi)^{\mathsf{b}}; (\psi)^{\mathsf{b}}))
\end{array}$$

- Case $l_E((v,v')) = \sqcup$
  $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\langle \gamma \sqcup \delta \rangle \varphi)^{\mathsf{b}}; \Gamma')$ and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, \mathit{dup}(\Gamma; \Gamma'; (\langle \gamma \rangle \varphi \vee \langle \delta \rangle \varphi)^{\mathsf{b}}))$

$$\begin{array}{rcl}
\mathsf{c}(\Gamma; (\langle \gamma \sqcup \delta \rangle \varphi)^{\mathsf{b}}; \Gamma') & = & \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\langle \gamma \sqcup \delta \rangle \varphi) \quad \text{(By Def. 6.2)} \\
& = & \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\langle \gamma \rangle \varphi \vee \langle \delta \rangle \varphi) + 1 \quad \text{(By Def. 6.1)} \\
& > & \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\langle \gamma \rangle \varphi \vee \langle \delta \rangle \varphi) \\
& = & \mathsf{c}(\Gamma; \Gamma'; (\langle \gamma \rangle \varphi \vee \langle \delta \rangle \varphi)^{\mathsf{b}}) \quad \text{(By Def. 6.2)} \\
& \geq & \mathsf{c}(\mathit{dup}(\Gamma; \Gamma'; (\langle \gamma \rangle \varphi \vee \langle \delta \rangle \varphi)^{\mathsf{b}})) \quad \text{(Given } \mathsf{c}(\mathit{dup}(\Gamma)) \leq \mathsf{c}(\Gamma)\text{)} \\
\\
\Rightarrow \mathsf{c}(\Gamma; (\langle \gamma \sqcup \delta \rangle \varphi)^{\mathsf{b}}; \Gamma') & > & \mathsf{c}(\mathit{dup}(\Gamma; \Gamma'; (\langle \gamma \rangle \varphi \vee \langle \delta \rangle \varphi)^{\mathsf{b}}))
\end{array}$$

- Case $l_E((v,v')) = ;$
  $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\langle \gamma ; \delta \rangle \varphi)^{\mathsf{b}}; \Gamma')$ and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, \mathit{dup}(\Gamma; \Gamma'; (\langle \gamma \rangle \langle \delta \rangle \varphi)^{\mathsf{b}}))$

$$\begin{array}{rcl}
\mathsf{c}(\Gamma; (\langle \gamma ; \delta \rangle \varphi)^{\mathsf{b}}; \Gamma') & = & \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\langle \gamma ; \delta \rangle \varphi) \quad \text{(By Def. 6.2)} \\
& = & \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\langle \gamma \rangle \langle \delta \rangle \varphi) + 1 \quad \text{(By Def. 6.1)} \\
& > & \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\langle \gamma \rangle \langle \delta \rangle \varphi) \\
& = & \mathsf{c}(\Gamma; \Gamma'; (\langle \gamma \rangle \langle \delta \rangle \varphi)^{\mathsf{b}}) \quad \text{(By Def. 6.2)} \\
& \geq & \mathsf{c}(\mathit{dup}(\Gamma; \Gamma'; (\langle \gamma \rangle \langle \delta \rangle \varphi)^{\mathsf{b}})) \quad \text{(Given } \mathsf{c}(\mathit{dup}(\Gamma)) \leq \mathsf{c}(\Gamma)\text{)} \\
\\
\Rightarrow \mathsf{c}(\Gamma; (\langle \gamma ; \delta \rangle \varphi)^{\mathsf{b}}; \Gamma') & > & \mathsf{c}(\mathit{dup}(\Gamma; \Gamma'; (\langle \gamma \rangle \langle \delta \rangle \varphi)^{\mathsf{b}}))
\end{array}$$

- Case $l_E((v,v')) = ?$
  $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\langle \psi ? \rangle \varphi)^{\mathsf{b}}; \Gamma')$ and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, \mathit{dup}(\Gamma; \Gamma'; (\psi \wedge \varphi)^{\mathsf{b}}))$

$$\begin{array}{rcl}
\mathsf{c}(\Gamma; (\langle \psi ? \rangle \varphi)^{\mathsf{b}}; \Gamma') & = & \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\langle \psi ? \rangle \varphi) \quad \text{(By Def. 6.2)} \\
& = & \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\psi \wedge \varphi) + 1 \quad \text{(By Def. 6.1)} \\
& > & \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\psi \wedge \varphi) \\
& = & \mathsf{c}(\Gamma; \Gamma'; (\psi \wedge \varphi)^{\mathsf{b}}) \quad \text{(By Def. 6.2)} \\
& \geq & \mathsf{c}(\mathit{dup}(\Gamma; \Gamma'; (\psi \wedge \varphi)^{\mathsf{b}})) \quad \text{(Given } \mathsf{c}(\mathit{dup}(\Gamma)) \leq \mathsf{c}(\Gamma)\text{)} \\
\\
\Rightarrow \mathsf{c}(\Gamma; (\langle \psi ? \rangle \varphi)^{\mathsf{b}}; \Gamma') & > & \mathsf{c}(\mathit{dup}(\Gamma; \Gamma'; (\psi \wedge \varphi)^{\mathsf{b}}))
\end{array}$$

- Case $l_E((v,v')) = !$

$l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\langle \psi! \rangle \varphi)^{\mathsf{b}}; \Gamma')$ and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, dup(\Gamma; \Gamma'; (\psi \vee \varphi)^{\mathsf{b}}))$

$$\begin{aligned}
\mathsf{c}(\Gamma; (\langle \psi! \rangle \varphi)^{\mathsf{b}}; \Gamma') &= \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\langle \psi! \rangle \varphi) \quad \text{(By Def. 6.2)} \\
&= \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\psi \vee \varphi) + 1 \quad \text{(By Def. 6.1)} \\
&> \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\psi \vee \varphi) \\
&= \mathsf{c}(\Gamma; \Gamma'; (\psi \vee \varphi)^{\mathsf{b}}) \quad \text{(By Def. 6.2)} \\
&\geq \mathsf{c}(dup(\Gamma; \Gamma'; (\psi \vee \varphi)^{\mathsf{b}})) \quad \text{(Given } \mathsf{c}(dup(\Gamma)) \leq \mathsf{c}(\Gamma))
\end{aligned}$$

$$\Rightarrow \mathsf{c}(\Gamma; (\langle \psi! \rangle \varphi)^{\mathsf{b}}; \Gamma') > \mathsf{c}(dup(\Gamma; \Gamma'; (\psi \vee \varphi)^{\mathsf{b}}))$$

- Case $l_E((v, v')) = \sqcap$
  $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\langle \gamma \sqcap \delta \rangle \varphi)^{\mathsf{b}}; \Gamma')$ and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, dup(\Gamma; \Gamma'; (\langle \gamma \rangle \varphi \wedge \langle \delta \rangle \varphi)^{\mathsf{b}}))$

$$\begin{aligned}
\mathsf{c}(\Gamma; (\langle \gamma \sqcap \delta \rangle \varphi)^{\mathsf{b}}; \Gamma') &= \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\langle \gamma \sqcap \delta \rangle \varphi) \quad \text{(By Def. 6.2)} \\
&= \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\langle \gamma \rangle \varphi \wedge \langle \delta \rangle \varphi) + 1 \quad \text{(By Def. 6.1)} \\
&> \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\langle \gamma \rangle \varphi \wedge \langle \delta \rangle \varphi) \\
&= \mathsf{c}(\Gamma; \Gamma'; (\langle \gamma \rangle \varphi \wedge \langle \delta \rangle \varphi)^{\mathsf{b}}) \quad \text{(By Def. 6.2)} \\
&\geq \mathsf{c}(dup(\Gamma; \Gamma'; (\langle \gamma \rangle \varphi \wedge \langle \delta \rangle \varphi)^{\mathsf{b}})) \quad \text{(Given } \mathsf{c}(dup(\Gamma)) \leq \mathsf{c}(\Gamma))
\end{aligned}$$

$$\Rightarrow \mathsf{c}(\Gamma; (\langle \gamma \sqcap \delta \rangle \varphi)^{\mathsf{b}}; \Gamma') > \mathsf{c}(dup(\Gamma; \Gamma'; (\langle \gamma \rangle \varphi \wedge \langle \delta \rangle \varphi)^{\mathsf{b}}))$$

- Case $l_E((v, v')) = \mathsf{mod}_m$
  $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\langle a \rangle \varphi)^{\mathsf{b}}; \Gamma'; (\langle a^d \rangle \psi)^{\mathsf{c}}; \Gamma'')$ and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, (\varphi)^{\mathsf{b}}; (\psi)^{\mathsf{c}})$

$$\begin{aligned}
\mathsf{c}(\Gamma; (\langle a \rangle \varphi)^{\mathsf{b}}; \Gamma'; (\langle a^d \rangle \psi)^{\mathsf{c}}; \Gamma'') &= \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\Gamma'') + \mathsf{c}(\langle a \rangle \varphi) + \mathsf{c}(\langle a^d \rangle \psi) \quad \text{(By Def. 6.2)} \\
&= \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\Gamma'') + \mathsf{c}(\varphi) + 1 + \mathsf{c}(\psi) + 1 \quad \text{(By Def. 6.1)} \\
&> \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\Gamma'') + \mathsf{c}(\varphi) + \mathsf{c}(\psi) \\
&= \mathsf{c}(\Gamma; \Gamma'; \Gamma''; (\varphi)^{\mathsf{b}}; (\psi)^{\mathsf{c}}) \quad \text{(By Def. 6.2)} \\
&\geq \mathsf{c}(dup(\Gamma; \Gamma'; \Gamma''; (\varphi)^{\mathsf{b}}; (\psi)^{\mathsf{c}})) \quad \text{(Given } \mathsf{c}(dup(\Gamma)) \leq \mathsf{c}(\Gamma))
\end{aligned}$$

$$\Rightarrow \mathsf{c}(\Gamma; (\langle a \rangle \varphi)^{\mathsf{b}}; \Gamma'; (\langle a^d \rangle \psi)^{\mathsf{c}}; \Gamma'') > \mathsf{c}(dup(\Gamma; \Gamma'; \Gamma''; (\varphi)^{\mathsf{b}}; (\psi)^{\mathsf{c}}))$$

  or $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\langle a^d \rangle \varphi)^{\mathsf{b}}; \Gamma'; (\langle a \rangle \psi)^{\mathsf{c}}; \Gamma'')$ and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, (\varphi)^{\mathsf{b}}; (\psi)^{\mathsf{c}})$

$$\begin{aligned}
\mathsf{c}(\Gamma; (\langle a^d \rangle \varphi)^{\mathsf{b}}; \Gamma'; (\langle a \rangle \psi)^{\mathsf{c}}; \Gamma'') &= \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\Gamma'') + \mathsf{c}(\langle a^d \rangle \varphi) + \mathsf{c}(\langle a \rangle \psi) \quad \text{(By Def. 6.2)} \\
&= \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\Gamma'') + \mathsf{c}(\varphi) + 1 + \mathsf{c}(\psi) + 1 \quad \text{(By Def. 6.1)} \\
&> \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\Gamma'') + \mathsf{c}(\varphi) + \mathsf{c}(\psi) \\
&= \mathsf{c}(\Gamma; \Gamma'; \Gamma''; (\varphi)^{\mathsf{b}}; (\psi)^{\mathsf{c}}) \quad \text{(By Def. 6.2)} \\
&\geq \mathsf{c}(dup(\Gamma; \Gamma'; \Gamma''; (\varphi)^{\mathsf{b}}; (\psi)^{\mathsf{c}})) \quad \text{(Given } \mathsf{c}(dup(\Gamma)) \leq \mathsf{c}(\Gamma))
\end{aligned}$$

$$\Rightarrow \mathsf{c}(\Gamma; (\langle a^d \rangle \varphi)^{\mathsf{b}}; \Gamma'; (\langle a \rangle \psi)^{\mathsf{c}}; \Gamma'') > \mathsf{c}(dup(\Gamma; \Gamma'; \Gamma''; (\varphi)^{\mathsf{b}}; (\psi)^{\mathsf{c}}))$$

Cases where $\exists v'' \in V : (v, v''), (v'', v') \in E$, $l_E((v, v'')) = \wedge$, and $l_E((v'', v')) \in \{\mathsf{left}, \mathsf{right}\}$:

- Case $l_E((v'', v')) = \mathsf{left}$:

$l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\varphi \wedge \psi)^{\mathsf{b}}; \Gamma')$ and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, \mathit{dup}(\Gamma; \Gamma'; (\varphi)^{\mathsf{b}}))$

$$\begin{aligned}
\mathsf{c}(\Gamma; (\varphi \wedge \psi)^{\mathsf{b}}; \Gamma') &= \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\varphi \wedge \psi) \quad \text{(By Def. 6.2)}\\
&= \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\varphi) + \mathsf{c}(\psi) + 1 \quad \text{(By Def. 6.1)}\\
&> \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\varphi) + \mathsf{c}(\psi)\\
&> \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\varphi) \quad \text{(Given } \mathsf{c}(\varphi) > 0\text{)}\\
&= \mathsf{c}(\Gamma; \Gamma'; (\varphi)^{\mathsf{b}}) \quad \text{(By Def. 6.2)}\\
&\geq \mathsf{c}(\mathit{dup}(\Gamma; \Gamma'; (\varphi)^{\mathsf{b}})) \quad \text{(Given } \mathsf{c}(\mathit{dup}(\Gamma)) \leq \mathsf{c}(\Gamma)\text{)}\\
\\
\Rightarrow \mathsf{c}(\Gamma; (\varphi \wedge \psi)^{\mathsf{b}}; \Gamma') &> \mathsf{c}(\mathit{dup}(\Gamma; \Gamma'; (\varphi)^{\mathsf{b}}))
\end{aligned}$$

- Case $l_E((v'', v')) = \mathsf{right}$
  $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\varphi \wedge \psi)^{\mathsf{b}}; \Gamma')$ and $l_V(v') = (N_V, \mathfrak{h}, \mathcal{H}, \mathit{dup}(\Gamma; \Gamma'; (\psi)^{\mathsf{b}}))$

$$\begin{aligned}
\mathsf{c}(\Gamma; (\varphi \wedge \psi)^{\mathsf{b}}; \Gamma') &= \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\varphi \wedge \psi) \quad \text{(By Def. 6.2)}\\
&= \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\varphi) + \mathsf{c}(\psi) + 1 \quad \text{(By Def. 6.1)}\\
&> \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\varphi) + \mathsf{c}(\psi)\\
&> \mathsf{c}(\Gamma) + \mathsf{c}(\Gamma') + \mathsf{c}(\psi) \quad \text{(Given } \mathsf{c}(\varphi) > 0\text{)}\\
&= \mathsf{c}(\Gamma; \Gamma'; (\psi)^{\mathsf{b}}) \quad \text{(By Def. 6.2)}\\
&\geq \mathsf{c}(\mathit{dup}(\Gamma; \Gamma'; (\psi)^{\mathsf{b}})) \quad \text{(Given } \mathsf{c}(\mathit{dup}(\Gamma)) \leq \mathsf{c}(\Gamma)\text{)}\\
\\
\Rightarrow \mathsf{c}(\Gamma; (\varphi \wedge \psi)^{\mathsf{b}}; \Gamma') &> \mathsf{c}(\mathit{dup}(\Gamma; \Gamma'; (\psi)^{\mathsf{b}}))
\end{aligned}$$

□

6.2 Finite Closure

A key property of CloG and SCloG for the termination condition is that there are finitely many formulas that can appear in sequents. This is shown using the following closure.

Definition 6.4. The closure $cl(\Phi)$ of a set of formulas $\Phi \subseteq \mathcal{L}_{\mathsf{NF}}$ is the smallest subset of $\mathcal{L}_{\mathsf{NF}}$ that contains Φ and is closed under the following rules:

$$\begin{array}{rll}
\text{If } \langle a\rangle\varphi \in cl(\Phi) \text{ or } \langle a^d\rangle\varphi \in cl(\Phi) & \text{then} & \varphi \in cl(\Phi)\\
\text{If } \varphi \vee \psi \in cl(\Phi) & \text{then} & \varphi, \psi \in cl(\Phi)\\
\text{If } \varphi \wedge \psi \in cl(\Phi) & \text{then} & \varphi, \psi \in cl(\Phi)\\
\text{If } \langle \gamma \sqcup \delta\rangle\varphi \in cl(\Phi) & \text{then} & \langle\gamma\rangle\varphi, \langle\delta\rangle\varphi \in cl(\Phi)\\
\text{If } \langle \gamma \sqcap \delta\rangle\varphi \in cl(\Phi) & \text{then} & \langle\gamma\rangle\varphi, \langle\delta\rangle\varphi \in cl(\Phi)\\
\text{If } \langle \gamma ; \delta\rangle\varphi \in cl(\Phi) & \text{then} & \langle\gamma\rangle\langle\delta\rangle\varphi \in cl(\Phi)\\
\text{If } \langle \psi ?\rangle\varphi \in cl(\Phi) & \text{then} & \psi \wedge \varphi \in cl(\Phi)\\
\text{If } \langle \psi !\rangle\varphi \in cl(\Phi) & \text{then} & \psi \vee \varphi \in cl(\Phi)\\
\text{If } \langle \gamma^*\rangle\varphi \in cl(\Phi) & \text{then} & \varphi \vee \langle\gamma\rangle\langle\gamma^*\rangle\varphi \in cl(\Phi)\\
\text{If } \langle \gamma^\times\rangle\varphi \in cl(\Phi) & \text{then} & \varphi \wedge \langle\gamma\rangle\langle\gamma^\times\rangle\varphi \in cl(\Phi)
\end{array}$$

Lemma 6.2. The size of the closure $|cl(\Gamma)|$ for a given set of formulas Γ is finite.

Lemma 6.3. Let $T = (V, E, l_V, l_E)$ be a well formed SCloG search tree for Γ_0. It must be that $|\{\Phi_\Gamma \mid v \in V, l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma)\}|$, which contains all sets of formulas in SCloG sequents in the tree, is bounded by $2^{|cl(\Phi_{\Gamma_0})|}$ and therefore is also finite.

Proof. If T is a well formed SCloG search tree for Γ_0 then the closure $cl(\Phi_{\Gamma_0})$ contains all of the formulas that can appear in the proof without annotations. This can be verified based on Definitions 5.10 and 5.11. This closure is also finite as no rules create a new arbitrary sub-formula. All possible set of formulas withing a sequent of SCloG must be in the power-set of the closure $\mathcal{P}(cl(\Phi_{\Gamma_0}))$ which has the size $2^{|cl(\Phi_{\Gamma_0})|}$. □

6.3 Finitely Many Applications of clo and $*$

Assumption 1. Given a branch in a well formed SCloG search tree $T = (V, E, l_V, l_E)$, if there are infinitely many pairs of vertices $v, v' \in V$ in the branch such that $(v, v') \in E$ and $l_E((v, v'))$ has the form clo_x, and all of the vertices v are matching unfoldings as defined in Definition 5.8. Then at least one vertex v is a cycling unfolding.

This assumption is made due to the fact that a correct definition has not yet been found to detect cycling unfoldings amongst matching unfoldings. It is suspected that there will be an occurrence of a cycling unfolding of some kind in an infinitely extending chain of matching unfoldings, as discussed in Section 4.5, but it may prove that a different termination condition is required.

Lemma 6.4. Given a branch in a well formed SCloG search tree $T = (V, E, l_V, l_E)$, there are finitely many pairs of vertices $v, v' \in V$ in the branch such that $(v, v') \in E$ and $l_E((v, v'))$ has the form clo_x

Proof. Suppose there is a branch $B = (V^B, E^B, l_V^B, l_E^B)$ in a well formed SCloG search tree $T = (V, E, l_V, l_E)$, such that there are infinitely many pairs of vertices $v, v' \in V^B$ such that $(v, v') \in E^B$ and $l_E((v, v'))$ has the form clo_x with $\mathsf{x} \in N$. Let V_{Clo} be the set of vertices $\{v \mid v \in V^B, \exists v' \in V^B : (v, v') \in E^B \ \& \ l_E((v, v')) = \mathsf{clo}_\mathsf{x}\}$. In this case $|V_{\mathsf{Clo}}| = \infty$. Then $\forall v \in V_{\mathsf{Clo}} : l_V^B(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma; (\langle\gamma\rangle\varphi)^\mathsf{a}; \Gamma')$. From Lemma 6.3, the possible values for the pair $(\langle\gamma\rangle\varphi, \Phi_{\Gamma;(\langle\gamma\rangle\varphi)^\mathsf{a};\Gamma'})$ is finite. From this, it must be that there is some value of $(\langle\gamma\rangle\varphi, \Phi_{\Gamma;(\langle\gamma\rangle\varphi)^\mathsf{a};\Gamma'})$ for which there are infinitely many vertices $v \in V_{\mathsf{Clo}}$ with $l_V^B(v) = (N_V', \mathfrak{h}', \mathcal{H}', \Psi; (\langle\gamma\rangle\varphi)^\mathsf{b}; \Psi')$ and $(\langle\gamma\rangle\varphi, \Phi_{\Psi;(\langle\gamma\rangle\varphi)^\mathsf{b};\Psi'}) = (\langle\gamma\rangle\varphi, \Phi_{\Gamma;(\langle\gamma\rangle\varphi)^\mathsf{a};\Gamma'})$. This means that the branch has infinitely many matching unfoldings from the Definition 5.8. According to Assumption 1, one of this infinitely extending chain of matching unfoldings is a cycling unfolding. According to Definition 5.10, a cycling unfolding can have no outgoing edges. This means that the branch ends at the occurrence of a cycling unfolding the there are not infinitely many vertices on the branch. Therefore, it can be concluded that there cannot be a branch with infinitely many pairs of vertices $v, v' \in V$ in the branch such that $(v, v') \in E$ and $l_E((v, v'))$ has the form clo_x □

Lemma 6.5. Given a branch in a well formed SCloG search tree $T = (V, E, l_V, l_E)$, there are finitely many pairs of vertices $v, v' \in V$ in the branch such that $(v, v') \in E$ and $l_E((v, v')) = *$

Proof. From Lemma 6.4, there are finitely many names introduced on a given branch. Let this set of names on a branch be N_b. If both the set of formulas in a well formed SCloG search tree for Γ_0, $cl(\Phi_{\Gamma_0})$, and the set of names N_b are finite. Then the set of annotated formulas on the branch $cl(\Phi_{\Gamma_0}) \times N_b$ is also finite. Each application of the SCloG rule $*$ must add a combination of an annotated formula and sequent to the set $\mathcal{H}$ that does not already occur in $\mathcal{H}$. There are finite possible annotated formulas and therefore finite sequents and pairs of sequents and formulas. So, on one branch there must be finite applications of $*$. □

6.4 Finite Search Tree

Lemma 6.6. Given a branch on a well formed SCloG search tree $T = (V, E, l_V, l_E)$, the length of any branch is finite.

Proof. Let $B = (V^B, E^B, l_V^B, l_E^B)$ be a branch of a well formed SCloG search tree $T = (V, E, l_V, l_E)$ where $|V^B| = \infty$. From Lemmas 6.4 and 6.5, there are finite applications of clo and $*$ on the branch. So after a certain point, there will be no more applications of the unfolding rules and only applications of the reduction rules for the rest of the infinitely extending branch. From Lemma 6.1, all reduction rules strictly reduce the complexity of the sequent in the successor. It is the case that the complexity of any sequent is greater than 0 so infinitely descending complexity is not possible. This contradiction means that the initial assumptions are not possible so $|V^B| \neq \infty$. □

Lemma 6.7. The size of well formed SCloG search tree $T = (V, E, l_V, l_E)$ is finite.

Proof. Let $T = (V, E, l_V, l_E)$ be a well formed SCloG search tree. Let $v \in V$ be a vertex such that $l_V(v) = (N_V, \mathfrak{h}, \mathcal{H}, \Gamma)$. The sequent Γ contains finitely many annotated formulas. All SCloG search rules in Definition 5.10 have active formulas that are of forms such that two different rules cannot share the same active formulas in a sequent. Therefore, there are finitely many different SCloG rules applicable to one sequent, and subsequently there are a finite number of branches coming from one vertex. Each branch also ends in only one leaf so there are finitely many leaves at the end of each branch. From König's Lemma [13], this makes a well formed SCloG search tree a finitely branching tree. Each search branch is then also finite in length according to Lemma 6.6. With no infinitely extending branches, an SCloG Search tree has no infinite path and so, by König's Lemma again, is finite. □

7 Towards Completeness Relative to CloG

In this section an argument is given for the completeness of SCloG relative to CloG. A complete proof is not given for completeness but an argument will be given to support the decisions made to reduce the search space of CloG to the rules of SCloG. Several lemmas are proven that work towards a proof that given a CloG proof exists for a root CloG-sequent Φ_0, then this can be transformed into a successful proof branch from a well formed SCloG search tree T for Γ_0 where $\Phi_{\Gamma_0} = \Phi_0$.

7.1 Simple Rule Reordering

Contrary to CloG, SCloG rules can only be applied in a specific order, and rules must be applied if they can be. Therefore, it must be shown that for all SCloG search rules with an ordering constraint, the associated CloG rules can be reordered from any place in a CloG proof to the position required to obtain a well formed SCloG search tree branch.

Lemma 7.1. If there is a CloG proof segment of the following form where $\star$ and $\diamond$ are simple rules as discussed in Section 4.3:

$$\dfrac{\dfrac{\Phi_3}{\Phi_2}\,\star}{\Phi_1}\,\diamond \quad \text{or} \quad \dfrac{\dfrac{\Phi_3 \qquad \Phi_3'}{\Phi_2}\,\star}{\Phi_1}\,\diamond \quad \text{or} \quad \dfrac{\dfrac{\Phi_3}{\Phi_2}\,\star \qquad \dfrac{\Phi_3'}{\Phi_2'}\,\star}{\Phi_1}\,\wedge \quad \text{or} \quad \dfrac{\dfrac{\Phi_3 \qquad \Phi_3'}{\Phi_2}\,\star \qquad \dfrac{\Phi_4 \qquad \Phi_4'}{\Phi_2'}\,\star}{\Phi_1}\,\diamond$$

Then if the rule $\star$ is applicable to Φ_1, the applications of $\star$ and $\diamond$ can be swapped.

Proof. The lemma is first shown for all cases of $\star$ where $\diamond$ is not $\wedge$:

- Case: $\star = \vee$
  Then the following CloG segment transformation is possible:

$$\dfrac{\dfrac{\Phi_2', (\varphi)^{\mathsf{a}}, (\psi)^{\mathsf{a}}}{\Phi_2', (\varphi \vee \psi)^{\mathsf{a}}}\,\vee}{\Phi_1', (\varphi \vee \psi)^{\mathsf{a}}}\,\diamond \quad \Rightarrow \quad \dfrac{\dfrac{\Phi_2', (\varphi)^{\mathsf{a}}, (\psi)^{\mathsf{a}}}{\Phi_1', (\varphi)^{\mathsf{a}}, (\psi)^{\mathsf{a}}}\,\diamond}{\Phi_1', (\varphi \vee \psi)^{\mathsf{a}}}\,\vee$$

- Case: $\star = \wedge$
  Then the following CloG segment transformation is possible:

$$\dfrac{\dfrac{\Phi_2', (\varphi)^{\mathsf{a}} \qquad \Phi_2', (\psi)^{\mathsf{a}}}{\Phi_2', (\varphi \wedge \psi)^{\mathsf{a}}}\,\wedge}{\Phi_1', (\varphi \wedge \psi)^{\mathsf{a}}}\,\diamond \quad \Rightarrow \quad \dfrac{\dfrac{\Phi_2', (\varphi)^{\mathsf{a}}}{\Phi_1', (\varphi)^{\mathsf{a}}}\,\diamond \qquad \dfrac{\Phi_2', (\psi)^{\mathsf{a}}}{\Phi_1', (\psi)^{\mathsf{a}}}\,\diamond}{\Phi_1', (\varphi \wedge \psi)^{\mathsf{a}}}\,\wedge$$

- Case: $\star = \sqcup$
  Then the following CloG segment transformation is possible:

$$\dfrac{\dfrac{\Phi_2', (\langle\gamma\rangle\varphi \vee \langle\delta\rangle\varphi)^{\mathsf{a}}}{\Phi_2', (\langle\gamma \sqcup \delta\rangle\varphi)^{\mathsf{a}}}\,\sqcup}{\Phi_1', (\langle\gamma \sqcup \delta\rangle\varphi)^{\mathsf{a}}}\,\diamond \quad \Rightarrow \quad \dfrac{\dfrac{\Phi_2', (\langle\gamma\rangle\varphi \vee \langle\delta\rangle\varphi)^{\mathsf{a}}}{\Phi_1', (\langle\gamma\rangle\varphi \vee \langle\delta\rangle\varphi)^{\mathsf{a}}}\,\diamond}{\Phi_1', (\langle\gamma \sqcup \delta\rangle\varphi)^{\mathsf{a}}}\,\sqcup$$

- Case: $\star = \sqcap$
  Then the following CloG segment transformation is possible:

$$\dfrac{\dfrac{\Phi_2', (\langle\gamma\rangle\varphi \wedge \langle\delta\rangle\varphi)^{\mathsf{a}}}{\Phi_2', (\langle\gamma \sqcap \delta\rangle\varphi)^{\mathsf{a}}}\,\sqcap}{\Phi_1', (\langle\gamma \sqcap \delta\rangle\varphi)^{\mathsf{a}}}\,\diamond \quad \Rightarrow \quad \dfrac{\dfrac{\Phi_2', (\langle\gamma\rangle\varphi \wedge \langle\delta\rangle\varphi)^{\mathsf{a}}}{\Phi_1', (\langle\gamma\rangle\varphi \wedge \langle\delta\rangle\varphi)^{\mathsf{a}}}\,\diamond}{\Phi_1', (\langle\gamma \sqcap \delta\rangle\varphi)^{\mathsf{a}}}\,\sqcap$$

- Case: $\star =$,
  Then the following CloG segment transformation is possible:

$$\dfrac{\dfrac{\Phi'_2, (\langle\gamma\rangle\langle\delta\rangle\varphi)^{\mathsf{a}}}{\Phi'_2, (\langle\gamma;\delta\rangle\varphi)^{\mathsf{a}}}\;;}{\Phi'_1, (\langle\gamma;\delta\rangle\varphi)^{\mathsf{a}}}\;\diamond \quad\Rightarrow\quad \dfrac{\dfrac{\Phi'_2, (\langle\gamma\rangle\langle\delta\rangle\varphi)^{\mathsf{a}}}{\Phi'_1, (\langle\gamma\rangle\langle\delta\rangle\varphi)^{\mathsf{a}}}\;\diamond}{\Phi'_1, (\langle\gamma;\delta\rangle\varphi)^{\mathsf{a}}}\;;$$

- Case: $\star = ?$
  Then the following CloG segment transformation is possible:

$$\dfrac{\dfrac{\Phi'_2, (\psi\wedge\varphi)^{\mathsf{a}}}{\Phi'_2, (\langle\psi?\rangle\varphi)^{\mathsf{a}}}\;?}{\Phi'_1, (\langle\psi?\rangle\varphi)^{\mathsf{a}}}\;\diamond \quad\Rightarrow\quad \dfrac{\dfrac{\Phi'_2, (\psi\wedge\varphi)^{\mathsf{a}}}{\Phi'_1, (\psi\wedge\varphi)^{\mathsf{a}}}\;\diamond}{\Phi'_1, (\langle\psi?\rangle\varphi)^{\mathsf{a}}}\;?$$

- Case: $\star = !$
  Then the following CloG segment transformation is possible:

$$\dfrac{\dfrac{\Phi'_2, (\psi\vee\varphi)^{\mathsf{a}}}{\Phi'_2, (\langle\psi!\rangle\varphi)^{\mathsf{a}}}\;!}{\Phi'_1, (\langle\psi!\rangle\varphi)^{\mathsf{a}}}\;\diamond \quad\Rightarrow\quad \dfrac{\dfrac{\Phi'_2, (\psi\vee\varphi)^{\mathsf{a}}}{\Phi'_1, (\psi\vee\varphi)^{\mathsf{a}}}\;\diamond}{\Phi'_1, (\langle\psi!\rangle\varphi)^{\mathsf{a}}}\;!$$

- Case: $\star = \ *$
  Then the following CloG segment transformation is possible:

$$\dfrac{\dfrac{\Phi'_2, (\varphi\vee\langle\gamma\rangle\langle\gamma^*\rangle\varphi)^{\mathsf{a}}}{\Phi'_2, (\langle\gamma^*\rangle\varphi)^{\mathsf{a}}}\;*}{\Phi'_1, (\langle\gamma^*\rangle\varphi)^{\mathsf{a}}}\;\diamond \quad\Rightarrow\quad \dfrac{\dfrac{\Phi'_2, (\varphi\vee\langle\gamma\rangle\langle\gamma^*\rangle\varphi)^{\mathsf{a}}}{\Phi'_1, (\varphi\vee\langle\gamma\rangle\langle\gamma^*\rangle\varphi)^{\mathsf{a}}}\;\diamond}{\Phi'_1, (\langle\gamma^*\rangle\varphi)^{\mathsf{a}}}\;*$$

- Case: $\star = \ \times$
  Then the following CloG segment transformation is possible:

$$\dfrac{\dfrac{\Phi'_2, (\varphi\wedge\langle\gamma\rangle\langle\gamma^\times\rangle\varphi)^{\mathsf{a}}}{\Phi'_2, (\langle\gamma^\times\rangle\varphi)^{\mathsf{a}}}\;\times}{\Phi'_1, (\langle\gamma^\times\rangle\varphi)^{\mathsf{a}}}\;\diamond \quad\Rightarrow\quad \dfrac{\dfrac{\Phi'_2, (\varphi\wedge\langle\gamma\rangle\langle\gamma^\times\rangle\varphi)^{\mathsf{a}}}{\Phi'_1, (\varphi\wedge\langle\gamma\rangle\langle\gamma^\times\rangle\varphi)^{\mathsf{a}}}\;\diamond}{\Phi'_1, (\langle\gamma^\times\rangle\varphi)^{\mathsf{a}}}\;\times$$

Then in the case that $\star$ is not $\wedge$ and $\diamond = \wedge$, the following CloG segment transformation is possible:

$$\dfrac{\dfrac{\Phi_2, (\varphi)^{\mathsf{a}}}{\Phi_1, (\varphi)^{\mathsf{a}}}\;\star \qquad \dfrac{\Phi_2, (\psi)^{\mathsf{a}}}{\Phi_1, (\psi)^{\mathsf{a}}}\;\star}{\Phi_1, (\varphi\wedge\psi)^{\mathsf{a}}}\;\wedge \quad\Rightarrow\quad \dfrac{\dfrac{\Phi_2, (\varphi)^{\mathsf{a}} \qquad \Phi_2, (\psi)^{\mathsf{a}}}{\Phi_2, (\varphi\wedge\psi)^{\mathsf{a}}}\;\wedge}{\Phi_1, (\varphi\wedge\psi)^{\mathsf{a}}}\;\star$$

In the last case that $\star = \diamond = \wedge$, the following CloG segment transformation is possible:

$$\dfrac{\dfrac{\Phi, (\varphi)^{\mathsf{a}}, (\varphi')^{\mathsf{b}} \qquad \Phi, (\psi)^{\mathsf{a}}, (\varphi')^{\mathsf{b}}}{\Phi, (\varphi\wedge\psi)^{\mathsf{a}}, (\varphi')^{\mathsf{b}}} \qquad \dfrac{\Phi, (\varphi)^{\mathsf{a}}, (\psi')^{\mathsf{b}} \qquad \Phi, (\psi)^{\mathsf{a}}, (\psi')^{\mathsf{b}}}{\Phi, (\varphi\wedge\psi)^{\mathsf{a}}, (\psi')^{\mathsf{b}}}\;\wedge}{\Phi, (\varphi\wedge\psi)^{\mathsf{a}}, (\varphi'\wedge\psi')^{\mathsf{b}}}\;\wedge \quad\Rightarrow$$

$$\dfrac{\dfrac{\Phi, (\varphi)^{\mathsf{a}}, (\varphi')^{\mathsf{b}} \qquad \Phi, (\varphi)^{\mathsf{a}}, (\psi')^{\mathsf{b}}}{\Phi, (\varphi)^{\mathsf{a}}, (\varphi'\wedge\psi')^{\mathsf{b}}} \qquad \dfrac{\Phi, (\psi)^{\mathsf{a}}, (\varphi')^{\mathsf{b}} \qquad \Phi, (\psi)^{\mathsf{a}}, (\psi')^{\mathsf{b}}}{\Phi, (\psi)^{\mathsf{a}}, (\varphi'\wedge\psi')^{\mathsf{b}}}\;\wedge}{\Phi, (\varphi\wedge\psi)^{\mathsf{a}}, (\varphi'\wedge\psi')^{\mathsf{b}}}\;\wedge$$

□

This lemma means that the simple rules can be rearranged to fit the ordering required by SCloG. Showing this supports the completeness relative to CloG of the strict ordering of rules in SCloG.

7.2 Structural Rules

The structural rules weak and exp are not rules in SCloG but are part of other SCloG search rules. Therefore it must be shown that if one of these structural rules is applied in a CloG proof, it can either be removed or associated with another rule, axiom, or closure and moved to fit that application.

Assumption 2. If there is an application of weak to an annotated formula $(\varphi)^{\mathsf{a}}$ in a CloG proof, then there are three possible occurrences in branches above the rule application where $(\varphi)^{\mathsf{a}}$ cannot be added to the sequent. These are applications of mod_m, Ax1 applications, or discharged $\mathsf{clo}_{\mathsf{x}}$ applications.

This assumption is based on the fact that these are the only parts of CloG that cannot have arbitrary side formulas, but it is not proven that these are the only cases affected by weak.

Lemma 7.2. If there is an application of weak to an annotated formula $(\varphi)^{\mathsf{a}}$ in a CloG proof, then the application of weak can be moved up to the closest application of either mod_m, Ax1, or discharged $\mathsf{clo}_{\mathsf{x}}$.

Proof. A well formed proof branch in CloG containing an application of weak will have the following form where $\bigtriangledown$ is an arbitrary CloG proof branch:

$$\begin{array}{c}\bigtriangledown\\ \dfrac{\Phi}{\Phi, (\varphi)^{\mathsf{a}}}\ \mathsf{weak}\end{array}$$

- Case where the branch $\bigtriangledown$ contains an application of mod_m (The proof branches $\bigtriangledown_1$ and $\bigtriangledown_2$ with the mod_m application combine to form $\bigtriangledown$, $\bigtriangledown_1$ does not contain an application of mod_m, and $\bigtriangledown_1'$ is the same as $\bigtriangledown_1$ but with φ^{a} appended to every sequent):

$$\begin{array}{c}\bigtriangledown_2\\ \dfrac{\psi^{\mathsf{b}}, \chi^{\mathsf{c}}}{(\langle a\rangle\psi)^{\mathsf{b}}, (\langle a^d\rangle\chi)^{\mathsf{c}}}\ \mathsf{mod}_m\\ \bigtriangledown_1\\ \dfrac{\Phi}{\Phi, \varphi^{\mathsf{a}}}\ \mathsf{weak}\end{array} \quad\Rightarrow\quad \begin{array}{c}\bigtriangledown_2\\ \dfrac{\dfrac{\psi^{\mathsf{b}}, \chi^{\mathsf{c}}}{(\langle a\rangle\psi)^{\mathsf{b}}, (\langle a^d\rangle\chi)^{\mathsf{c}}}\ \mathsf{mod}_m}{(\langle a\rangle\psi)^{\mathsf{b}}, (\langle a^d\rangle\chi)^{\mathsf{c}}, \varphi^{\mathsf{a}}}\ \mathsf{weak}\\ \bigtriangledown_1'\\ \Phi, \varphi^{\mathsf{a}}\end{array}$$

- Case where $\bigtriangledown$ contains no application of mod_m and ends in Ax1 (The combination of the branch $\bigtriangledown_1$ and Ax1 application form $\bigtriangledown$, and $\bigtriangledown_1'$ is the same as $\bigtriangledown_1$ but with φ^{a} appended to every sequent):

$$\begin{array}{c}\dfrac{}{p^{\epsilon}, (\neg p)^{\epsilon}}\ \mathsf{Ax1}\\ \bigtriangledown_1\\ \dfrac{\Phi}{\Phi, (\varphi)^{\mathsf{a}}}\ \mathsf{weak}\end{array} \quad\Rightarrow\quad \begin{array}{c}\dfrac{\dfrac{}{p^{\epsilon}, (\neg p)^{\epsilon}}\ \mathsf{Ax1}}{p^{\epsilon}, (\neg p)^{\epsilon}, \varphi^{\mathsf{a}}}\ \mathsf{weak}\\ \bigtriangledown_1'\\ \Phi, \varphi^{\mathsf{a}}\end{array}$$

- Case where $\bigtriangledown$ contains no application of mod_m and ends in a discharged $\mathsf{clo}_{\mathsf{x}}$ application:
  - Case where $\bigtriangledown$ contains the application of $\mathsf{clo}_{\mathsf{x}}$ (The combination of the branches $\bigtriangledown_1, \bigtriangledown_2$ with the $\mathsf{clo}_{\mathsf{x}}$ application and closure make up $\bigtriangledown$, $\bigtriangledown_1'$ and $\bigtriangledown_2'$ are the same as $\bigtriangledown_1$ and $\bigtriangledown_2$ but with φ^{a} appended to every sequent):

$$\begin{array}{c} [\Phi_3]^{\mathsf{x}} \\ \nabla_2 \\ \dfrac{\Phi_2}{\Phi_1}\,\mathsf{clo_x} \\ \nabla_1 \\ \dfrac{\Phi_0}{\Phi_0, (\varphi)^{\mathsf{a}}}\,\mathsf{weak} \end{array} \quad \Rightarrow \quad \begin{array}{c} [\Phi_3, (\varphi)^{\mathsf{a}}]^{\mathsf{x}} \\ \nabla_2' \\ \dfrac{\Phi_2, (\varphi)^{\mathsf{a}}}{\Phi_1, (\varphi)^{\mathsf{a}}}\,\mathsf{clo_x} \\ \nabla_1' \\ \Phi_0, (\varphi)^{\mathsf{a}} \end{array}$$

- Case where ∇ does not contain the application of $\mathsf{clo_x}$ (The combination of the branch ∇_1 with the discharged $\mathsf{clo_x}$ make up ∇, ∇_1' is the same as ∇_1 but with φ^{a} appended to every sequent):

$$\begin{array}{c} [\Phi']^{\mathsf{x}} \\ \nabla_1 \\ \dfrac{\Phi}{\Phi, (\varphi)^{\mathsf{a}}}\,\mathsf{weak} \end{array} \quad \Rightarrow \quad \begin{array}{c} \dfrac{[\Phi']^{\mathsf{x}}}{\Phi', (\varphi)^{\mathsf{a}}}\,\mathsf{weak} \\ \nabla_1' \\ \Phi, (\varphi)^{\mathsf{a}} \end{array}$$

□

Assumption 3. If there is an application of exp to an annotated formula $(\varphi)^{\mathsf{axb}}$ in a CloG proof, then there are two possible occurrences in branches above the rule application where formulas with the annotation ab could not be changed to axb and exp must be moved there. These are applications of $\mathsf{Ax1}$ or discharged $\mathsf{clo_x}$ applications. There are no other cases where exp must be applied.

This assumption is based on the parts of CloG that are dependent on labels, and are the only cases accounted for in SCloG. There is also a case where weak removes the annotation ab so will also remove axb.

Lemma 7.3. If there is an application of exp to an annotated formula $(\varphi)^{\mathsf{axb}}$ in a CloG proof, then the application of exp can be moved up to the closest application of either $\mathsf{Ax1}$, discharged $\mathsf{clo_x}$, or removed due to an application of weak.

Proof. A well formed CloG proof branch which contains an application of exp will have the following form where ∇ is an arbitrary CloG proof branch:

$$\begin{array}{c} \nabla \\ \dfrac{\Phi, (\varphi)^{\mathsf{ab}}}{\Phi, (\varphi)^{\mathsf{axb}}}\,\mathsf{exp} \end{array}$$

- Case where ∇ contains an application of weak that removes the only formula with the annotation ab (The branches ∇_1 and ∇_2 with the weak application combine to make up the ∇, ∇_1' is the same as ∇_1 but with annotations of ab replaced with axb):

$$\begin{array}{c} \nabla_2 \\ \forall(\chi)^{\mathsf{c}} \in \Phi' : \mathsf{c} \neq \mathsf{ab}\ \dfrac{\Phi'}{\Phi', \psi^{\mathsf{ab}}}\,\mathsf{weak} \\ \nabla_1 \\ \dfrac{\Phi, \varphi^{\mathsf{ab}}}{\Phi, \varphi^{\mathsf{axb}}}\,\mathsf{exp} \end{array} \quad \Rightarrow \quad \begin{array}{c} \nabla_2 \\ \forall(\chi)^{\mathsf{c}} \in \Phi' : \mathsf{c} \neq \mathsf{axb}\ \dfrac{\Phi'}{\Phi', \psi^{\mathsf{axb}}}\,\mathsf{weak} \\ \nabla_1' \\ \Phi, \varphi^{\mathsf{axb}} \end{array}$$

- Case where $\bigtriangledown$ contains no applications of weak that remove the only formula with the annotation ab and ends in $\mathsf{Ax1}$ (The branch $\bigtriangledown_1$ and $\mathsf{Ax1}$ make up $\bigtriangledown$, $\bigtriangledown'_1$ is the same as $\bigtriangledown_1$ but with annotations of ab replaced with axb):

$$\begin{array}{c} \dfrac{}{p^{\epsilon}, (\neg p)^{\epsilon}}\ \mathsf{Ax1} \\ \bigtriangledown_1 \\ \dfrac{\Phi, \varphi^{\mathsf{ab}}}{\Phi, \varphi^{\mathsf{axb}}}\ \mathsf{exp} \end{array} \quad \Rightarrow \quad (\mathsf{c}, \mathsf{d}) \in \{(\mathsf{x}, \epsilon), (\epsilon, \mathsf{x}), (\mathsf{x}, \mathsf{x})\} \begin{array}{c} \dfrac{\dfrac{}{p^{\epsilon}, (\neg p)^{\epsilon}}\ \mathsf{Ax1}}{p^{\mathsf{c}}, (\neg p)^{\mathsf{d}}}\ \mathsf{exp}^* \\ \bigtriangledown'_1 \\ \Phi, \varphi^{\mathsf{axb}} \end{array}$$

- Case where $\bigtriangledown$ contains no applications of weak that remove the only formula with the annotation ab and ends in a discharged $\mathsf{clo}_{\mathsf{x}}$ application (The branch $\bigtriangledown_1$ and discharged clo_x application make up $\bigtriangledown$, $\bigtriangledown'_1$ is the same as $\bigtriangledown_1$ but with annotations of ab replaced with axb, and Φ'' is the same as Φ' but with zero or more occurrences of x that are not in Φ'):

$$\begin{array}{c} [\Phi']^{\mathsf{y}} \\ \bigtriangledown_1 \\ \dfrac{\Phi, \varphi^{\mathsf{ab}}}{\Phi, \varphi^{\mathsf{axb}}}\ \mathsf{exp} \end{array} \quad \Rightarrow \quad \begin{array}{c} \dfrac{[\Phi']^{\mathsf{y}}}{\Phi''}\ \mathsf{exp}{*} \\ \bigtriangledown'_1 \\ \Phi, \varphi^{\mathsf{axb}} \end{array}$$

□

7.3 Limited Applications of $*$

The SCloG search rule $*$ cannot be applied again to the same fixed point formula in the same sequent. This is not a condition of the rule in CloG, so there may be an instance of the CloG rule $*$ being applied more than once to the same fixed point and sequent in the given CloG proof. In that case, the sub proof from that sequent after the last application can replace that of the first application and this will still result in a well formed CloG proof. This step can be taken first before the rest of the transformation. So given a well formed CloG proof segment of the following form (where $\bigtriangledown$ is an arbitrary CloG proof segment).

Lemma 7.4. If there is a well formed CloG proof branch where the same sequent appears more than once, the proof segment between the first occurrence and last occurrence of the sequent can be removed to still obtain a well formed CloG proof.

Proof. A proof branch with multiple occurrences of the same sequent can be transformed in the following way:

$$\begin{array}{c} \dfrac{\bigtriangledown}{\Phi}\ * \\ \vdots \\ \dfrac{}{\Phi}\ * \end{array} \quad \Rightarrow \quad \dfrac{\bigtriangledown}{\Phi}\ *$$

□

8 Implementation and Testing

8.1 Rascal Project Structure

The implementation is split into three packages, “CloG_Base”, “Proof_Search” and “Bound_Calculation”. The “CloG_Base” module contains the concrete and abstract syntax types for Game Logic and CloG inherited from previous bachelor theses [6, 7, 8]. It also contains the LaTeX output module and main module “CloG_Base::GLTool” used to interact with the tool on the command line. Most functions in the “GlTool” module are used for testing but the main function “findProofFromSequent” will output a latex file with the solved proof given the name of an input sequent file, or will output no proof on the command line if none exists. The “Bound_Calculation” module is not used in the proof search algorithm but was used to investigate the closure $cl()$ and boundedness of a CloG proof tree. The algorithm itself is then implemented in the “Proof_Search” package.

8.2 Algorithm Implementation

The “Proof_Search” package contains three modules, “Util”, “RuleApplications”, and “ProofSearch”. The “Util” module contains some extra types and aliases for the proof search such as the fixed point unfolding recording which contains a fixed point and a set of annotated formulas. The “Util” module also has the function to remove non-discharged clo applications and a function to check the partial order $\preceq$. The “RuleApplications” module mainly contains the functions “canApply” and “Apply” which check for applicable rules and generate the next sequent from a rule application. Lastly, the “ProofSearch” contains the recursive proof search function and the main function which calls the recursive function followed by the clo removal post-processing step. The complete Rascal code can be found in Appendix A.

8.3 Manual Test Strategy

Various provable and non-provable sequents were tested manually with the proof solver. A list of the sequents and the resulting output proof (or no proof if none was found) is given below:

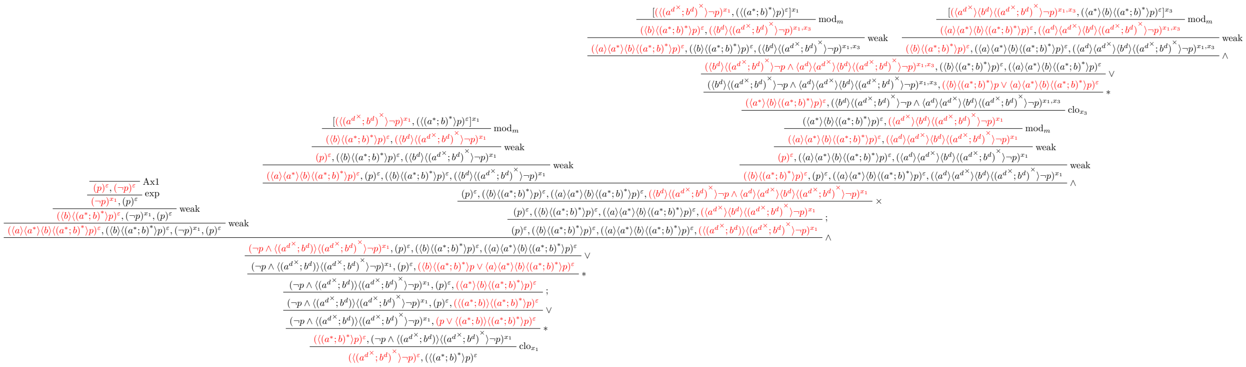
- $(\langle a \sqcap b\rangle p \vee \langle a^d \sqcup b^d\rangle \neg p)^\epsilon$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(p)^\varepsilon, (\neg p)^\varepsilon}\,\text{Ax1}}{(\langle a\rangle p)^\varepsilon, (\langle a^d\rangle \neg p)^\varepsilon}\,\text{mod}_m}{(\langle b^d\rangle \neg p)^\varepsilon, (\langle a\rangle p)^\varepsilon, (\langle a^d\rangle \neg p)^\varepsilon}\,\text{weak} \qquad \dfrac{\dfrac{\dfrac{}{(p)^\varepsilon, (\neg p)^\varepsilon}\,\text{Ax1}}{(\langle b\rangle p)^\varepsilon, (\langle b^d\rangle \neg p)^\varepsilon}\,\text{mod}_m}{(\langle a^d\rangle \neg p)^\varepsilon, (\langle b\rangle p)^\varepsilon, (\langle b^d\rangle \neg p)^\varepsilon}\,\text{weak}}{\dfrac{(\langle a^d\rangle \neg p)^\varepsilon, (\langle b^d\rangle \neg p)^\varepsilon, (\langle a\rangle p \wedge \langle b\rangle p)^\varepsilon}{\dfrac{(\langle (a \sqcap b)\rangle p)^\varepsilon, (\langle a^d\rangle \neg p)^\varepsilon, (\langle b^d\rangle \neg p)^\varepsilon}{\dfrac{(\langle (a \sqcap b)\rangle p)^\varepsilon, (\langle a^d\rangle \neg p \vee \langle b^d\rangle \neg p)^\varepsilon}{\dfrac{(\langle (a \sqcap b)\rangle p)^\varepsilon, (\langle (a^d \sqcup b^d)\rangle \neg p)^\varepsilon}{(\langle (a \sqcap b)\rangle p \vee \langle (a^d \sqcup b^d)\rangle \neg p)^\varepsilon}\,\vee}\,\sqcup}\,\vee}\,\sqcap}\,\wedge$$

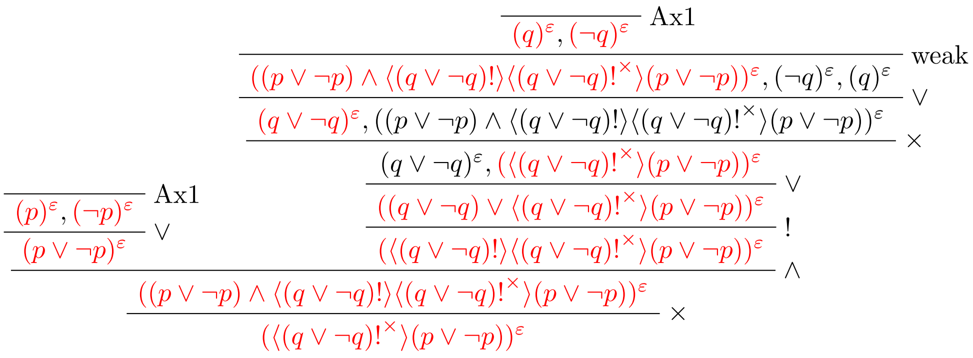
- $(\langle a^{d^\times}\rangle \neg p \vee \langle a^*\rangle p)^\epsilon$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(p)^\varepsilon, (\neg p)^\varepsilon}\,\text{Ax1}}{(\neg p)^{x_1}, (p)^\varepsilon}\,\text{exp}}{(\langle a\rangle\langle a^*\rangle p)^\varepsilon, (\neg p)^{x_1}, (p)^\varepsilon}\,\text{weak} \qquad \dfrac{\dfrac{[(\langle a^{d^\times}\rangle \neg p)^{x_1}, (\langle a^*\rangle p)^\varepsilon]^{x_1}}{(\langle a\rangle\langle a^*\rangle p)^\varepsilon, (\langle a^d\rangle\langle a^{d^\times}\rangle \neg p)^{x_1}}\,\text{mod}_m}{(p)^\varepsilon, (\langle a\rangle\langle a^*\rangle p)^\varepsilon, (\langle a^d\rangle\langle a^{d^\times}\rangle \neg p)^{x_1}}\,\text{weak}}{\dfrac{(\neg p \wedge \langle a^d\rangle\langle a^{d^\times}\rangle \neg p)^{x_1}, (p)^\varepsilon, (\langle a\rangle\langle a^*\rangle p)^\varepsilon}{\dfrac{(\neg p \wedge \langle a^d\rangle\langle a^{d^\times}\rangle \neg p)^{x_1}, (p \vee \langle a\rangle\langle a^*\rangle p)^\varepsilon}{\dfrac{(\langle a^*\rangle p)^\varepsilon, (\neg p \wedge \langle a^d\rangle\langle a^{d^\times}\rangle \neg p)^{x_1}}{\dfrac{(\langle a^{d^\times}\rangle \neg p)^\varepsilon, (\langle a^*\rangle p)^\varepsilon}{(\langle a^{d^\times}\rangle \neg p \vee \langle a^*\rangle p)^\varepsilon}\,\vee}\,\text{clo}_{x_1}}\,*}\,\vee}\,\wedge$$

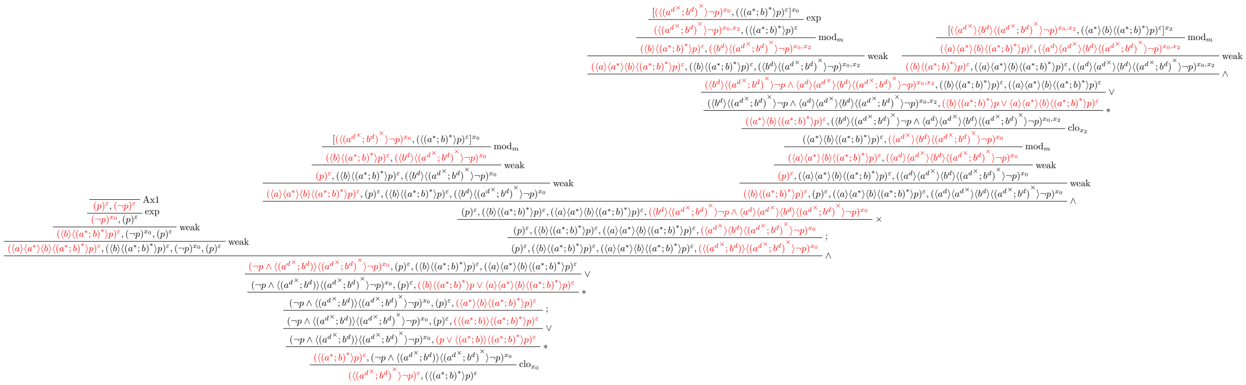
- $(\langle (a^{d^{\times}}; b^d)^{\times} \rangle \neg p)^{\varepsilon}, (\langle (a^*; b)^* \rangle p)^{\varepsilon}$:



- $(\langle b^{d^*} \rangle \langle c^{\times} \rangle p)^{\epsilon}, (\langle b^{\times} \rangle q)^{\epsilon}$: no proof found
- $(\langle (q \vee \neg q)!^{\times} \rangle (p \vee \neg p))^{\epsilon}$:


$(q)^\varepsilon, (\neg q)^\varepsilon$ Ax1
$((p \vee \neg p) \wedge \langle (q \vee \neg q)! \rangle \langle (q \vee \neg q)!^{\times} \rangle (p \vee \neg p))^\varepsilon, (\neg q)^\varepsilon, (q)^\varepsilon$ weak
$(q \vee \neg q)^\varepsilon, ((p \vee \neg p) \wedge \langle (q \vee \neg q)! \rangle \langle (q \vee \neg q)!^{\times} \rangle (p \vee \neg p))^\varepsilon$ $\vee$
$(q \vee \neg q)^\varepsilon, (\langle (q \vee \neg q)!^{\times} \rangle (p \vee \neg p))^\varepsilon$ $\times$
$((q \vee \neg q) \vee \langle (q \vee \neg q)!^{\times} \rangle (p \vee \neg p))^\varepsilon$ $\vee$
$(\langle (q \vee \neg q)! \rangle \langle (q \vee \neg q)!^{\times} \rangle (p \vee \neg p))^\varepsilon$!
$(p)^\varepsilon, (\neg p)^\varepsilon$ Ax1
$(p \vee \neg p)^\varepsilon$ $\vee$
$((p \vee \neg p) \wedge \langle (q \vee \neg q)! \rangle \langle (q \vee \neg q)!^{\times} \rangle (p \vee \neg p))^\varepsilon$ $\wedge$
$(\langle (q \vee \neg q)!^{\times} \rangle (p \vee \neg p))^\varepsilon$ $\times$

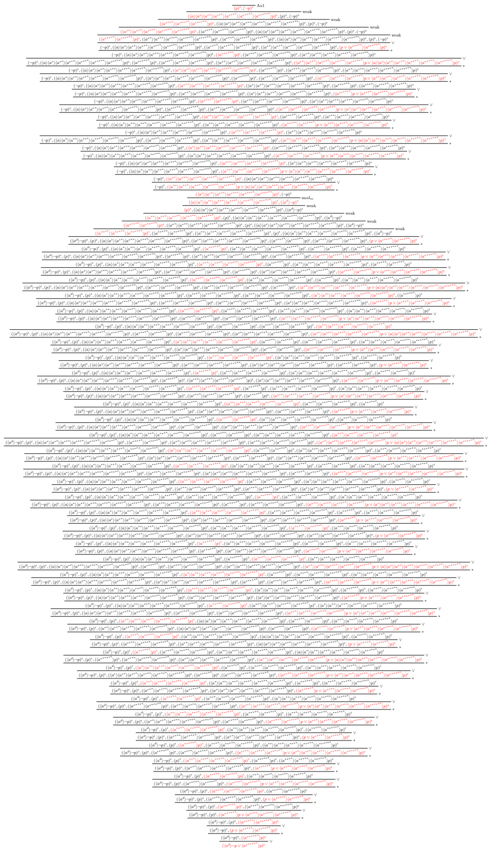
- $\langle ((a^{\times} \sqcup b) * \sqcup b^*)^{\times} \sqcap a^* \rangle p^{\epsilon}$: no proof found
- $\langle ((a^{\times} \sqcup b)^* \sqcup (a^* \sqcap b^*)^{\times})^{\times} \rangle p^{\epsilon}$: no proof found
- $\langle ((a^{\times} \sqcup b) * \sqcup b^*)^{\times} \sqcap a^* \rangle p^{\epsilon}, \langle ((a^{\times} \sqcup b)^* \sqcup (a^* \sqcap b^*)^{\times})^{\times} \rangle p^{\epsilon}$: no proof found
- $(\langle ((a^d)^{\times}; b^d)^{\times} \rangle \neg p)^{\epsilon} (\langle (a^*; b)^* \rangle p)^{\epsilon}$:



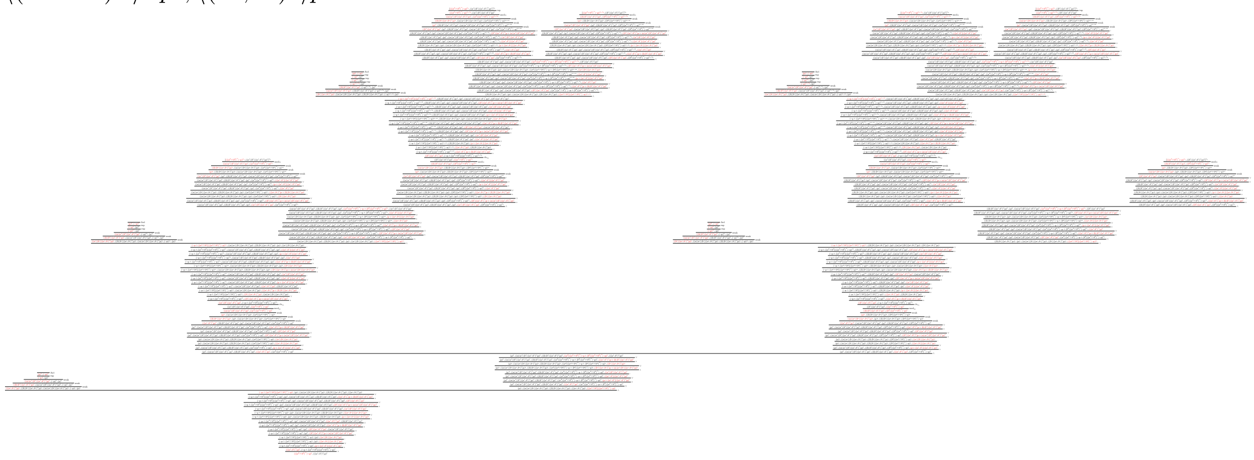
- $(\neg p \vee \langle a^{******} \rangle p)^{\epsilon}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(p)^{\varepsilon}, (\neg p)^{\varepsilon}}\,\text{Ax1}}{(\langle a^{*****}\rangle\langle a^{******}\rangle p)^{\varepsilon}, (p)^{\varepsilon}, (\neg p)^{\varepsilon}}\,\text{weak}}{(\neg p)^{\varepsilon}, (p \vee \langle a^{*****}\rangle\langle a^{******}\rangle p)^{\varepsilon}}\,\vee}{(\neg p)^{\varepsilon}, (\langle a^{******}\rangle p)^{\varepsilon}}\,*}{(\neg p \vee \langle a^{******}\rangle p)^{\varepsilon}}\,\vee$$

- $(\langle a^d\rangle\neg p \vee \langle a^{******}\rangle p)^{\epsilon}$:



- $\langle (a^d \sqcap b^d)^{\times}\rangle\neg p^{\epsilon}, \langle (a^*; b^*)^*\rangle p^{\epsilon}$:



9 Discussion and Conclusion

The aim of this study was to find a terminating proof search algorithm for the cyclic cut-free sequent calculus, CloG. This had been attempted previously in Meerholz [8] but with minimal concrete definitions and proofs to support the decisions made for the algorithm. Therefore, this project put a focus on the theoretical definition of the algorithm to make it provably sound and complete relative to CloG and terminating in all cases. This was achieved by strictly defining the search space for a depth first search algorithm in the form of the proof search system SCloG. The soundness of SCloG was shown by defining a transformation from a successful SCloG search branch to a well formed CloG proof with the same root sequent. For completeness relative to CloG, while a complete proof was not given, argumentation has been given for how each decision made for the algorithm should maintain completeness relative to CloG.

It was then proven that with finitely many applications of clo_x per branch, the algorithm must terminate. This result is accompanied by an investigation into a way to ensure finitely many applications of clo_x per branch, but this was not yet defined in an implementable fashion. The pursuit of this specific condition was highly motivated by the fact that it did not require the reduction of Game Logic to a smaller fragment.

The decision to order the simple rules in the way they are ordered in SCloG, in this case, is based mainly on intuition and from studying the optimal shape of CloG proofs. This may not lead to optimally shaped proofs but testing shows that by applying binary inference rules as late as possible, and by applying clo_x earlier where possible, the proofs will have a minimal shape in most cases. Currently the global checks made on the history of the branch, both for clo_x and for cycling unfolding are quite computationally costly. These costs are minimised by checking the computationally efficient conditions first in order to avoid making slower checks as often.

The definition of SCloG offers a strong base to optimise the search space of a CloG proof solver while also allowing concrete correctness proofs about the algorithm. This technique for search space definition can be further applied to other proof search algorithms outside of just Game Logic. By defining the search space in this way, branches of the proof search could be further explored beyond just the successful proof branches which may give further insight into how to optimise both this search algorithm and the formal proof systems themselves.

10 Future Work

The main questions left open by this project are the final condition of the termination proof and the proof of the completeness of SCloG relative to CloG. The last termination condition required is for there to be finitely many applications of $\mathsf{clo_x}$ on a branch. Although another termination condition may prove to be required if the cycling unfoldings cannot be identified. Finding more examples of matching $\mathsf{clo_x}$ applications will give a better understanding of what is possible amongst matching $\mathsf{clo_x}$ applications. Another direction that could be taken is to reduce the Game Logic formulas used in sequents for the algorithm to a "flat fragment" as mentioned in Section 3. This would put stronger constraints on the possibilities of fixed point unfoldings and how they interact with each other that will provide more initial conditions for a proof of termination, and also completeness relative to CloG.

While some partial results and observations are given, the completeness of SCloG relative to CloG should be formulated into a full proof. This could perhaps be done by defining a stricter transformation from a CloG proof to a successful SCloG branch. However, the completeness of CloG is currently uncertain as of the result found in [14], where the μ-calculus proof system Clo is shown to be incomplete due a specific valid sequent case for which a proof does not exist. As mentioned in Section 1, the completeness result in [5] is based on a transformation from Clo to CloG, which was at the time assumed to be complete. Game Logic is only a fragment of the modal μ-calculus so it may be that this exact case is outside of that fragment, but that result is yet to be investigated. So it may be beneficial to first confirm the completeness of CloG before extending this to SCloG.

Inspecting the search space may provide valuable insight into how to correct and then further optimise the algorithm. So an extension of the implementation to also be able to construct and output SCloG proof search trees, either in their entirety or just in segments, would make this inspection much easier as currently trees can only be drawn out by hand. As mentioned in Section 4.3, most of the rules have an arbitrary ordering because of not much being known about how they affect the size of proofs. Inspecting the search tree may make more information about this clear by comparing multiple different SCloG derivations in the same tree. Currently, the algorithm also does not utilise any heuristic techniques to decide which paths of the search tree to explore first. Aiming for a specific closure condition or axiom at a given point may help the algorithm reach successful branches faster in many cases, even if the worst case does not change. Analysing SCloG search trees may also lead to some better ideas for how to implement these heuristics as well.

References

[1] Marc Pauly and Rohit Parikh. Game logic - an overview. *Studia Logica*, 75:165–182, 11 2003. URL: https://link.springer.com/article/10.1023/A:1027354826364, doi:10.1023/A:1027354826364.

[2] Rohit Parikh. The logic of games and its applications. In *Selected Papers of the International Conference on "Foundations of Computation Theory" on Topics in the Theory of Computation*, page 111–139, USA, 1985. Elsevier North-Holland, Inc.

[3] Nicolas Troquard and Philippe Balbiani. Propositional Dynamic Logic. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2023 edition, 2023.

[4] Andrea Aler Tubella and Lutz Strassburger. Introduction to Deep Inference. Lecture, August 2019. URL: https://inria.hal.science/hal-02390267.

[5] Sebastian Enqvist, Helle Hvid Hansen, Clemens Kupke, Johannes Marti, and Yde Venema. Completeness for game logic, 2019. URL: https://arxiv.org/abs/1904.07691, doi:10.48550/ARXIV.1904.07691.

[6] Christopher Worthington. Proof transformations for game logic, 2021. URL: https://fse.studenttheses.ub.rug.nl/25673.

[7] Steven J. van Schagen. Game logic: A proof transformation from gentzen to hilbert, 2022. URL: https://fse.studenttheses.ub.rug.nl/28264.

[8] Han Meerholz. Towards automated theorem proving in the clog proof system, 2021. URL: https://fse.studenttheses.ub.rug.nl/id/eprint/28388.

[9] Bahareh Afshari and Graham E. Leigh. Cut-free completeness for modal mu-calculus. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005088.

[10] Dominik Wehr. An abstract framework for the analysis of cyclic derivations, 2021. URL: https://eprints.illc.uva.nl/id/eprint/1812.

[11] Johannes Marti and Yde Venema. Focus-style proof systems and interpolation for the alternation-free μ-calculus, 2021. URL: https://arxiv.org/abs/2103.01671, doi:10.48550/ARXIV.2103.01671.

[12] James Brotherston, Nikos Gorogiannis, and Rasmus L Petersen. A generic cyclic theorem prover. In *Asian Symposium on Programming Languages and Systems*, pages 350–367. Springer, 2012.

[13] König's lemma. URL: https://mathworld.wolfram.com/KoenigsLemma.html.

[14] Johannes Kloibhofer. A note on the incompleteness of afshari & leigh's system clo, 2023. arXiv:2307.06846.

A Complete Rascal Code

A.1 Package “CloG_Base”

Listing 1: ”CloGSyntax.rsc”

```
module CloG_Base::CloGSyntax

/*
 * Module defines the concrete syntax for the input of a CloG proof
 *
 * Converted to abstract syntax tree by CST2AST_CloG module
 */

extend lang::std::Layout;

// Proof file starts with "CloG" and wrapped with {}
start syntax SCloGProof
        = "CloG" "{" SCloGPart* ps "}";

/*
 * The CloG file consists of a list of sequents paired with either the rule they are inferred by, a discarg
 *
 * The list of sequents are also written in the order from the bottom of a proof tree to the top.
 *
 * For the use of the conjuction rule, the proof splits into two branches that are wrapped by two sets of
 */
syntax SCloGPart
        = SCloGSequent seq SCloGRule rule ";"
        | SCloGSequent seq "and" ";" "{" SCloGPart* psL "}" "{" SCloGPart* psR "}"
        | "leaf" ";";

/*
 * Each sequent consists of a list of CLoG terms wrapped by [] for which the syntax is also defined.
 */
start syntax SCloGSequent = "Seq" "[" SCloGTerm* terms "]";

/*
 * Each CloG term is defined as a CloG expression (a game logic formula in normal form) with a superscript
 *
 * The annotation is defined as a list of named Id's wrapped by [].
 */
syntax SCloGTerm
        = "\<" SCloGExpr ex "^" "[" SId* label "]" "\>"
        | SCloGExpr ex "^" "[" SId* label "]";

// Syntax definition for list of rules defined for CloG, excluding the conjunction rule
syntax SCloGRule
        = "Ax1"
        | "modm"
        | "or"
        | "choice"
        | "dChoice"
        | "weak"
        | "exp"
        | "concat"
        | "iter"
        | "dIter"
        | "test"
        | "dTest"
        | "clo" "_" SId n
        | "closure" "^" SId n;

// Syntax definition of normal form game logic expression used in CloG
syntax SCloGExpr
        = prop: SId p
        | not: "~" SId p
        > left par: "(" SCloGExpr ex ")"
        > right strat: "\<" SCloGGame g "\>" SCloGExpr ex
        > left and: SCloGExpr exL "&" SCloGExpr exR
        > left or: SCloGExpr exL "|" SCloGExpr exR;

// Syntax definition of normal form game expression used in CloG
syntax SCloGGame
        = agame: SId g
        | dual: SId g "^d"
        > left par: "(" SCloGGame ga ")"
        > left \test: SCloGExpr ex "?"
        > left dTest: SCloGExpr ex "!"
        > left iter: SCloGGame ga "*"
        > left dIter: SCloGGame ga "^x"
        > left con: SCloGGame gaL ";" SCloGGame gaR
        > left dChoice: SCloGGame gaL "&&" SCloGGame gaR
        > left choice: SCloGGame gaL "||" SCloGGame gaR;

// Syntax definition of a named ID which can have a subscript integer
// Used for atomic games, atomic propositions, and Clo rule names
syntax SId
        = Id n "_" Int sub
        | Id n;

// To avoid ambiguity with the underscore, we define an identifier as only consisting
// of letters (which is already more liberal than the single letter restraint in the
// literature.)
lexical Id = [a-z A-Z]+;

// Regular Expression definition for an integer that can be used for an ID subscript
lexical Int
        = [1-9][0-9]*
        | [0];
```

Listing 2: ”GLASTs.rsc”

```
module CloG_Base::GLASTs
/*
 * Module defining all Abstract Syntax Types for the Game logic proof transformations
 */

/*
 * Abstract Syntax for a CloG Proof
 */

data CloGProof(loc src = |tmp:///|)
        = CloGLeaf()
        | disClo(CloGSequent seq, CloGName n)
        | CloGUnaryInf(CloGSequent seq, CloGRule rule, CloGProof inf)
        | CloGBinaryInf(CloGSequent seq, CloGProof infL, CloGProof infR);

alias CloGSequent = list[CloGTerm];

data CloGTerm(loc src = |tmp:///|)
        = term(GameLog s, list[CloGName] label, bool active);

data CloGRule(loc src = |tmp:///|)
        = ax1()
        | modm()
        | andR()
        | orR()
        | choiceR()
        | dChoiceR()
        | weak()
        | exp()
        | concatR()
        | iterR()
        | testR()
        | dIterR()
        | dTestR()
        | clo(CloGName n);

data CloGName(loc src = |tmp:///|)
        = name(str l)
        | nameS(str l, int sub);

/*
 * Abstract Syntax for a Game Logic Formula
 */

data GameLog(loc src = |tmp:///|)
        = top()
        | bot()
        | atomP(Prop p)
        | neg(GameLog pr)
        | \mod(Game g, GameLog pr)
        | modExp(Game g, GameLog pr, int n) // fp annotation for expansion rule
        | dMod(Game g, GameLog pr)
        | and(GameLog pL, GameLog pR)
        | or(GameLog pL, GameLog pR)
        | cond(GameLog pL, GameLog pR)
        | biCond(GameLog pL, GameLog pR);

data Game(loc src = |tmp:///|)
        = atomG(AGame g)
        | dual(Game ga)
        | \test(GameLog p)
        | dTest(GameLog p)
        | iter(Game ga)
        | dIter(Game ga)
        | dIterExp(Game ga, int n) // fp annotation for expansion rule
        | concat(Game gL, Game gR)
        | choice(Game gL, Game gR)
        | dChoice(Game gL, Game gR);

/*
 * Abstract syntax for atomic games and propositions
 */

data AGame(loc src = |tmp:///|)
        = agame(str l)
        | agameS(str l, int sub);

data Prop(loc src = |tmp:///|)
        = prop(str l)
        | propS(str l, int sub);
```

Listing 3: ”CST2AST_CloG.rsc”

```
module CloG_Base::CST2AST_CloG
/*
 * Module containing function to transform CloG input to an abstract syntax tree
 */

import ParseTree;
import String;
import List;

import CloG_Base::CloGSyntax;
import CloG_Base::GLASTs;

// Same function name for conversion used throughout and functionality depends on type of input

/* Main function for syntax conversion
 *
 * Input: Parsed CloG input syntax
 * Output: CloGProof algebraic type for whole proof
 */
CloGProof cst2astCloG(start[SCloGProof] sp){
        SCloGProof p = sp.top;

        return cst2astCloG([ pp | SCloGPart pp ← p.ps ]);
}

/* Function for syntax conversion of a list of sequents and rules. Recursive function, identifies and conve
 * list using a switch then calls the same function on the remainder of the list.
 *
 * Input: list of concrete syntax proof parts
 * Output CloGProof algebraic type for the inputted parts
 */
CloGProof cst2astCloG(list[SCloGPart] ps){
        switch(head(ps)){
                case (SCloGPart)‘leaf ;‘:
                        return CloGLeaf();
                case (SCloGPart)‘<SCloGSequent seq> closure ^ <SId n> ;‘:
                        return disClo(cst2astCloG(seq),
                                                   id2name(n));
                case (SCloGPart)‘<SCloGSequent seq> <SCloGRule rule> ;‘:
                        return CloGUnaryInf(cst2astCloG(seq),
                                                                  cst2astCloG(rule),
                                                                  cst2astCloG(tail(ps)));
                case (SCloGPart)‘<SCloGSequent seq> and ; { <SCloGPart* psL> } { <SCloGPart* psR> }‘:
                        return CloGBinaryInf(cst2astCloG(seq),
                                                                  cst2astCloG([ pp | SCloGPart pp ←
psL ]),
                                                                  cst2astCloG([ pp | SCloGPart pp ←
psR ]));
                default: throw "Unsupported CloG Proof";
        }
}

/* Function for syntax conversion of a CloG sequent.
 *
 * Input: Concrete syntax CloG sequent
 * Output: CloG sequent algebraic type
 */
CloGSequent cst2astCloG((SCloGSequent)‘Seq [ <SCloGTerm* terms> ]‘) = [cst2astCloG(t) | SCloGTerm t ←
terms ];

/* Function for syntax conversion of a CloG term. Conversion called on the proposition and list reduction u
 *
 * Input: Concrete syntax CloG term
 * Output: CloG term algebraic type
 */
CloGTerm cst2astCloG((SCloGTerm)‘\< <SCloGExpr ex> ^ [ <SId* label> ] \>‘){
        return term(cst2astCloG(ex), [id2name(n) | SId n ← label], true);
}
CloGTerm cst2astCloG((SCloGTerm)‘<SCloGExpr ex> ^ [ <SId* label> ]‘){
        return term(cst2astCloG(ex), [id2name(n) | SId n ← label], false);
}

/* Function for syntax converstion of a CloG rule. Switch used to identify which rule is used.
 *
 * Input: Concrete syntax CloG rule
 * Output: CloG rule algebraic type
 */
CloGRule cst2astCloG(SCloGRule r){
        switch(r){
                case (SCloGRule)‘Ax1‘: return ax1();
                case (SCloGRule)‘modm‘: return modm();
                case (SCloGRule)‘or‘: return orR();
                case (SCloGRule)‘choice‘: return choiceR();
                case (SCloGRule)‘dChoice‘: return dChoiceR();
                case (SCloGRule)‘weak‘: return weak();
                case (SCloGRule)`exp`: return exp();
                case (SCloGRule)`concat`: return concatR();
                case (SCloGRule)`iter`: return iterR();
                case (SCloGRule)`dIter`: return dIterR();
                case (SCloGRule)`test`: return testR();
                case (SCloGRule)`dTest`: return dTestR();
                case (SCloGRule)`clo _ <SId n>`: return clo(id2name(n));
                default: throw "Unsupported CloG Rule";
        }
}

/* Function for syntax conversion of a normal form game logic propositional formula. Switch statement used
 * operator is used.
 *
 * Input: Concrete syntax game expression
 * Output: Game logic proposition algebraic type
 */
GameLog cst2astCloG(SCloGExpr exp){
        switch (exp){
                case(SCloGExpr)`~ <SId p>`:
                        return neg(atomP(id2prop(p)));
                case(SCloGExpr)`<SId p>`:
                        return atomP(id2prop(p));
                case(SCloGExpr)`( <SCloGExpr ex> )`:
                        return cst2astCloG(ex);
                case(SCloGExpr)`\< <SCloGGame g> \> <SCloGExpr ex>`:
                        return \mod(cst2astCloG(g), cst2astCloG(ex));
                case(SCloGExpr)`<SCloGExpr exL> & <SCloGExpr exR>`:
                        return and(cst2astCloG(exL), cst2astCloG(exR));
                case(SCloGExpr)`<SCloGExpr exL> | <SCloGExpr exR>`:
                        return or(cst2astCloG(exL), cst2astCloG(exR));
                default: throw "Unsupported Game Logic Formula";
        }
}

/* Function for syntax conversion of a normal form game formula. Switch statement used to identify which
 * operator is used.
 *
 * Input: Concrete syntax game expression
 * Output: Game algebraic type
 */
Game cst2astCloG(SCloGGame g){
        switch (g){
                case (SCloGGame)`<SId a> ^d`:
                        return dual(atomG(id2agame(a)));
                case (SCloGGame)`<SId a>`:
                        return atomG(id2agame(a));
                case (SCloGGame)`( <SCloGGame ga> )`:
                        return cst2astCloG(ga);
                case (SCloGGame)`<SCloGExpr ex> ?`:
                        return \test(cst2astCloG(ex));
                case (SCloGGame)`<SCloGExpr ex> !`:
                        return dTest(cst2astCloG(ex));
                case (SCloGGame)`<SCloGGame ga> *`:
                        return iter(cst2astCloG(ga));
                case (SCloGGame)`<SCloGGame ga> ^x`:
                        return dIter(cst2astCloG(ga));
                case (SCloGGame)`<SCloGGame gaL> ; <SCloGGame gaR>`:
                        return concat(cst2astCloG(gaL), cst2astCloG(gaR));
                case (SCloGGame)`<SCloGGame gaL> && <SCloGGame gaR>`:
                        return dChoice(cst2astCloG(gaL), cst2astCloG(gaR));
                case (SCloGGame)`<SCloGGame gaL> || <SCloGGame gaR>`:
                        return choice(cst2astCloG(gaL), cst2astCloG(gaR));
                default: throw "Unsupported Game Formula";
        }
}

/* ID conversion functions need different names as they use the same concrete syntax type. Is more modular
```

```
 * than a larger switch statement.
 *
 * Input: Concrete syntax for an identifier
 * Output: Algebraic the corresponding atom or name
 */

// Function for syntax conversion of a Clo rule name
CloGName id2name(SId n){
        switch (n){
                case (SId)`<Id id> _ <Int sub>`:
                        return nameS("<id>",toInt("<sub>"));
                case (SId)`<Id id>`:
                        return name("<id>");
                default: throw "Unsupported Name";
        }
}

// Function for syntax conversion of an atomic proposition
Prop id2prop(SId p){
        switch (p){
                case (SId)`<Id id> _ <Int sub>`:
                        return propS("<id>",toInt("<sub>"));
                case (SId)`<Id id>`:
                        return prop("<id>");
                default: throw "Unsupported Proposition";
        }
}

// Function for syntax conversion of an atomic game
AGame id2agame(SId a){
        switch (a){
                case (SId)`<Id id> _ <Int sub>`:
                        return agameS("<id>",toInt("<sub>"));
                case (SId)`<Id id>`:
                        return agame("<id>");
                default: throw "Unsupported Proposition";
        }
}
```

Listing 4: ”LaTeXOutput.rsc”

```
module CloG_Base::LaTeXOutput
/*
 * Module defining the transformation from abstract proof trees to LaTeX proof trees
 */

import CloG_Base::GLASTs;

import IO;
import List;
import Set;

// Functions takes a proof tree and file location as input then writes the LaTeX output to the file.
void LaTeXOutput(CloGProof p, loc out){
        writeFile(out, LaTeXOutput(p));
}

// Functions define the preamle and proof tree wrapper for LaTeX output
str LaTeXOutput(CloGProof p) =
        "\\documentclass{article}
        '\\usepackage[utf8]{inputenc}
        '\\usepackage{prooftrees}
        '\\usepackage{graphics}
        '\\usepackage{xcolor}
        '
        '\\begin{document}
        '
        '\\resizebox{\\textwidth}{!}{
```

```
        '<LaTeXCloGTree(p)>
        '\\DisplayProof
        '}
        '
        '\\end{document}";

// Function to output the LaTeX proof tree part for each CloG sequent and assocated rule label
str LaTeXCloGTree(CloGLeaf()) =
        "\\AxiomC{}";
str LaTeXCloGTree(disClo(CloGSequent seq, CloGName n)) =
        "\\AxiomC{$[<LaTeXCloGSequent(seq, false)>]^{<LaTeXCloGTree(n)>}$}";
str LaTeXCloGTree(CloGUnaryInf(CloGSequent seq, CloGRule rule, CloGProof inf)) =
        "<LaTeXCloGTree(inf)>
        '\\RightLabel{<LaTeXCloGTree(rule)>}
        '\\UnaryInfC{<LaTeXCloGSequent(seq, true)>}";
str LaTeXCloGTree(CloGBinaryInf(CloGSequent seq, CloGProof infL, CloGProof infR)) =
        "<LaTeXCloGTree(infL)>
        '<LaTeXCloGTree(infR)>
        '\\RightLabel{$\\wedge$}
        '\\BinaryInfC{<LaTeXCloGSequent(seq, true)>}";

// Function to output the superscript label attached to each logic formula and to
// specify the color of the term depending on whether it is active or not
str LaTeXCloGTree(term(GameLog s, [], bool active))
        = "<active ? "\\textcolor{red}{" : "">(<LaTeXGameLog(s)>)^{\\varepsilon}<active ? "}" :
"">";
str LaTeXCloGTree(term(GameLog s, list[CloGName] label, bool active))
        = "<active ? "\\textcolor{red}{" : "">(<LaTeXGameLog(s)>)^{<(LaTeXCloGTree(head(label)) | it
          + ", " + LaTeXCloGTree(n) | CloGName n ← tail(label))>} <active ? "}" : "">";

// Function to output the CloG rule labels
str LaTeXCloGTree(ax1()) = "Ax1";
str LaTeXCloGTree(modm()) = "mod$_{m}$";
str LaTeXCloGTree(andR()) = "$\\wedge$";
str LaTeXCloGTree(orR()) =  "$\\vee$";
str LaTeXCloGTree(choiceR()) =  "$\\sqcup$";
str LaTeXCloGTree(dChoiceR()) =  "$\\sqcap$";
str LaTeXCloGTree(weak()) =  "weak";
str LaTeXCloGTree(exp()) =  "exp";
str LaTeXCloGTree(concatR()) =  "$;$";
str LaTeXCloGTree(iterR()) =  "$*$";
str LaTeXCloGTree(testR()) =  "$?$";
str LaTeXCloGTree(dIterR()) =  "$\\times$";
str LaTeXCloGTree(dTestR()) =  "$!$";
str LaTeXCloGTree(clo(CloGName n)) =  "clo$_{<LaTeXCloGTree(n)>}$";

// Function to output a CloG name which can have a subscript
str LaTeXCloGTree(name(str n)) = "<n>";
str LaTeXCloGTree(nameS(str n, int sub)) = "<n>_{<sub>}";


str LaTeXCloGSequent([], false) = "";
str LaTeXCloGSequent([], false) = "$ $";
str LaTeXCloGSequent(list[CloGTerm] terms, false) = "<(LaTeXCloGTree(head(terms)) | it + ", " + LaTeXCloGTr
tail(terms))>";
str LaTeXCloGSequent(list[CloGTerm] terms, true) = "$<(LaTeXCloGTree(head(terms)) | it + ", " + LaTeXCloGTr
tail(terms))>$";

// Function to output the game logic formulae in LaTeX maths mode
str LaTeXGameLog(top()) = "(p\\vee\\neg p)";
str LaTeXGameLog(bot()) = "(p\\wedge\\neg p)";
str LaTeXGameLog(atomP(Prop p)) = "<LaTeXGameLog(p)>";
str LaTeXGameLog(neg(GameLog pr)) = "\\neg <hasPar(pr)>";
str LaTeXGameLog(\mod(Game g, GameLog pr)) = "\\langle <hasPar(g)>\\rangle <hasPar(pr)>";
str LaTeXGameLog(dMod(Game g, GameLog pr)) = "[<hasPar(g)>]<hasPar(pr)>";
str LaTeXGameLog(and(GameLog pL, GameLog pR)) = "<hasPar(pL)>\\wedge <hasPar(pR)>";
str LaTeXGameLog(or(GameLog pL, GameLog pR)) = "<hasPar(pL)>\\vee <hasPar(pR)>";
str LaTeXGameLog(cond(GameLog pL, GameLog pR)) = "<hasPar(pL)>\\rightarrow <hasPar(pR)>";
str LaTeXGameLog(biCond(GameLog pL, GameLog pR)) = "<hasPar(pL)>\\leftrightarrow <hasPar(pR)>";

// Function to output the game formulae in LaTeX maths mode
str LaTeXGameLog(atomG(AGame g)) = "<LaTeXGameLog(g)>";
str LaTeXGameLog(dual(Game ga)) = "{<hasPar(ga)>}^d";
str LaTeXGameLog(\test(GameLog g)) = "<hasPar(g)>?";
str LaTeXGameLog(dTest(GameLog g)) = "<hasPar(g)>!";
str LaTeXGameLog(iter(Game ga)) = "{<hasPar(ga)>}^{*}";
str LaTeXGameLog(dIter(Game ga)) = "{<hasPar(ga)>}^{\\times}";
str LaTeXGameLog(concat(Game gL, Game gR)) = "<hasPar(gL)>;<hasPar(gR)>";
str LaTeXGameLog(choice(Game gL, Game gR)) = "<hasPar(gL)>\\sqcup <hasPar(gR)>";
str LaTeXGameLog(dChoice(Game gL, Game gR)) = "<hasPar(gL)>\\sqcap <hasPar(gR)>";

// Function to put parentheses around only the binary connectives and not unary connectives or atomic formu
str hasPar(top()) = "(p\\vee\\neg p)";
str hasPar(bot()) = "(p\\wedge\\neg p)";
str hasPar(atomP(Prop p)) = "<LaTeXGameLog(atomP(p))>";
str hasPar(neg(GameLog pr)) = "<LaTeXGameLog(neg(pr))>";
str hasPar(\mod(Game g, GameLog pr)) = "<LaTeXGameLog(\mod(g,pr))>";
str hasPar(dMod(Game g, GameLog pr)) = "<LaTeXGameLog(dMod(g,pr))>";
str hasPar(and(GameLog pL, GameLog pR)) = "(<LaTeXGameLog(and(pL,pR))>)";
str hasPar(or(GameLog pL, GameLog pR)) = "(<LaTeXGameLog(or(pL,pR))>)";
str hasPar(cond(GameLog pL, GameLog pR)) = "(<LaTeXGameLog(cond(pL,pR))>)";
str hasPar(biCond(GameLog pL, GameLog pR)) = "(<LaTeXGameLog(biCond(pL,pR))>)";
str hasPar(atomG(AGame g)) = "<LaTeXGameLog(atomG(g))>";
str hasPar(dual(Game ga)) = "<LaTeXGameLog(dual(ga))>";
str hasPar(\test(GameLog g)) = "<LaTeXGameLog(\test(g))>";
str hasPar(dTest(GameLog g)) = "<LaTeXGameLog(dTest(g))>";
str hasPar(iter(Game ga)) = "<LaTeXGameLog(iter(ga))>";
str hasPar(dIter(Game ga)) = "<LaTeXGameLog(dIter(ga))>";
str hasPar(concat(Game gL, Game gR)) = "(<LaTeXGameLog(concat(gL,gR))>)";
str hasPar(choice(Game gL, Game gR)) = "(<LaTeXGameLog(choice(gL,gR))>)";
str hasPar(dChoice(Game gL, Game gR)) = "(<LaTeXGameLog(dChoice(gL,gR))>)";

// Functions to output the atomic propositions and games which can have a subscript
str LaTeXGameLog(agame(str n)) = "<n>";
str LaTeXGameLog(agameS(str n, int sub)) = "<n>_{<sub>}";
str LaTeXGameLog(prop(str n)) = "<n>";
str LaTeXGameLog(propS(str n, int sub)) = "<n>_{<sub>}";
```

Listing 5: ”GLTool.rsc”

```
module CloG_Base::GLTool
/*
 * Main module for proof search tool
 *
 * Call findProofFromSequent with the name of the input file to use.
 * Output .tex file can be found in the output folder.
 */

import CloG_Base::CloGSyntax;
import CloG_Base::GLASTs;
import CloG_Base::CST2AST_CloG;
import CloG_Base::LaTeXOutput;

import Proof_Search::ProofSearch;
import Proof_Search::Util;

import ParseTree;
import IO;

// // Main function is split up for modularity and to help with testing in the terminal.

// Get the location of an input .clog file.
loc inputLoc(str file) {
        return (|project://game-logic-automated-theorem-prover/input| + file)[extension=".clog"];
}
```

```
// Get the location of an input .seq file.
loc inputSeqLoc(str file) {
        return (|project://game-logic-automated-theorem-prover/input| + file)[extension=".seq"];
}

// Form the location for an output .tex file
loc outputLoc(str file) {
        return (|project://game-logic-automated-theorem-prover/output| + file)[extension=".tex"];
}

// Parse input .clog file and output abstract syntax tree for the CloG Proof
CloGProof getCloGAST(str file) {
        loc l = inputLoc(file);
        start[SCloGProof] cst = parse(#start[SCloGProof], l);
        return cst2astCloG(cst);
}

// Get a CloGSequent from a given input sequent file name
CloGSequent inputSequent(str file) {
        loc l = inputSeqLoc(file);
        start[SCloGSequent] seqCst = parse(#start[SCloGSequent], l);
        return [cst2astCloG(term) | SCloGTerm term ← seqCst.top.terms];
}

// Get the first term from a given input sequent file name
CloGTerm getTerm(str file) {
        loc l = inputSeqLoc(file);
        start[SCloGSequent] seqCst = parse(#start[SCloGSequent], l);
        return cst2astCloG(seqCst.top.terms[0]);
}

// Perform a proof search for a given sequent input file name and, if a proof exists, output the result wit
void findProofFromSequent(str \in) {
        CloGSequent sequent = inputSequent(\in);
        MaybeProof proof = findProof(sequent);
        if (!(noProof(p) := proof)) {
                CloG2LaTeX(proof.p, \in);
        } else {
                print("no proof found\n");
        }
}

// Input abstract syntax tree for CloG proof and output CloG .tex proof tree to the given output file
void CloG2LaTeX(CloGProof p, str out) {
        loc l = outputLoc(out);
        LaTeXOutput(p, l);
}

// Input .clog file name and output CloG output .tex proof tree to the given output file
void input2latex(str \in,  str out) {
        CloG2LaTeX(getCloGAST(\in), out);
}
```

A.2 Package “Proof_Search”

Listing 6: ”Util.rsc”

```
module Proof_Search::Util

import CloG_Base::GLASTs;
import Exception;
import util::Math;
import CloG_Base::LaTeXOutput;

import List;
import IO;
/*
 * A FpSeq, or fixpoint sequent is a tuple containing a sequent that contains
 * a specific fixpoint formula, and the index of the term in that sequent which
 * contains the fixpoint formula.
 */
alias FpSeq = tuple[set[CloGTerm] contextSeq, CloGTerm fpFormula];

/*
 * A CloSeqs is a map, mapping each name associated with a fixpoint formula to the
 * FpSeq containing that formula at the specified index.
 */
alias CloSeqs = map[CloGName name, FpSeq fpSeq];

/*
 * MaybeName is either a closeName containing the name for a clo name that can be discharged, a repFp which
 * the current sequent needs to be checked for a cycling unfolding, or noClose if there is no possible clos
 */
data MaybeName
        = closeName(CloGName x)
        | noClose();

/*
 * MaybeTerm is either a appTerm containing a term to which a rule can be applied, or noTerm() which indica
 * apply a rule to
 */
data MaybeTerm
        = appTerm(CloGTerm term)
        | appTerms(CloGTerm term1, CloGTerm term2);

/*
 * MaybeProof is either a CloGProof for a successful proof search branch or noProof(), which
 * indicates no proof could be found
 */
data MaybeProof
        = proof(CloGProof p)
        | proofN(CloGProof p, int n)
        | noProof(int n);

/*
 * A utility function to remove all labels and activity from a sequent to inspect
 * only the set of formulae.
 */
set[GameLog] emptyLabels(CloGSequent sequent) = {t.s | CloGTerm t ← sequent};

/*
 * A function that replaces closure rules that no not have
 * a discharged assumption by a dIter rule instead.
 *
 * Input:  A CloGProof that may contain closure rule applications
 *         without a discharged assumption appearing somewhere above
 *         the proof node at which the closure rule was applied
 * Output: A CloGProof with its closure rule applications without
 *         assumptions replaced by dIter applications
 *
 * For each of the nodes at which a closure rule was applied, and
 * above which no discharged assumption appears, the replaceCloAt()
 * function is called, which handles the actual replacement.
 */
CloGProof replaceUnusedClos(CloGProof proof) {
        return visit(proof) {
                case subProof:CloGUnaryInf(_, clo(CloGName n), CloGProof inf): {
                        if (/disClo(_, n) !:= inf) {
                                insert replaceCloAt(subProof, n);
                        }
                }
        }
}

/*
 * A function that replaces a specific closure rule application
 * in a proof by a dIter rule application, and updates the rest
 * of the proof accordingly
 *
 * Input:  A CloGProof with a closure rule application at the root,
 *         and the name associated with this closure rule application
 * Output: A CloGProof with the closure rule application at the root
 *         replaced by a dIter rule application, and no more closure
 *         rule name appearing in sequents above this proof node
 *
 * Replaces the rule at the root by a dIter rule, then visits the
 * rest of the proof, and for all sequents above the starting node,
 * the name associated with the original closure rule application
 * is removed.
 */
CloGProof replaceCloAt(CloGProof proof, CloGName name) {
        proof.rule = dIterR();

        proof = visit(proof) {
                case subP:CloGUnaryInf(CloGSequent seq, _, _): {
                        CloGSequent newSeq = [];
                        for (term ← seq) {
                                if (t:term(_, list[CloGName] label, _) := term) {
                                        t.label = label - name;
                                        newSeq += t;
                                }
                        }
                        subP.seq = newSeq;
                        insert subP;
                }
                case subP:CloGBinaryInf(CloGSequent seq, _, _): {
                        CloGSequent newSeq = [];
                        for (term ← seq) {
                                if (t:term(_, list[CloGName] label, _) := term) {
                                        t.label = label - name;
                                        newSeq += t;
                                }
                        }
                        subP.seq = newSeq;
                        insert subP;
                }
                case subP:disClo(CloGSequent seq, _): {
                        CloGSequent newSeq = [];
                        for (term ← seq) {
                                if (t:term(_, list[CloGName] label, _) := term) {
                                        t.label = label - name;
                                        newSeq += t;
                                }
                        }
                        subP.seq = newSeq;
                        insert subP;
                }
        }

        return visit(proof) {
                case CloGUnaryInf(CloGSequent seq, exp(), subP2): {
                        if (toSet(seq) == toSet(subP2.seq)) {
                                insert subP2;
                        }
                }
        }
}

/*
 * Generate new unused clo name of the form x_n
 */
CloGName getNewName(list[CloGName] names) {
        int newNumber = (-1 | max(it, \num) | nameS("x",int \num) ← names) + 1;
        return nameS("x", newNumber);
}

/*
 * An algorithm returning whether one fixpoint formula is less than or equal to
 * the other, according to the fixpoint ordering defined in the literature.
 *
 * Input:  two GameLog formulae
 * Output: a bool, true if the left GameLog formula is less than or equal to the
 *         right GameLog formula according to the fixpoint ordering, and false
 *         otherwise.
 *
 * One fixpoint formula is considered less than or equal to another fixpoint
 * formula, if the game in the other fixpoint formula is a subterm of the game
 * in the one fixpoint formula.
 */
bool fpLessThanOrEqualTo(GameLog left, GameLog right) {
        if (\mod(Game fp0, _) := left && \mod(Game fp1, _) := right){
                if (
                (iter(_) := fp0 || dIter(_) := fp0)
         && (iter(_) := fp1 || dIter(_) := fp1)
                ) {
                        ret = subTerm(fp0, fp1);
                        return ret;
                }
        }
        throw IllegalArgument("Error: cannot apply fixpoint ordering on non-fixpoint formulae!");
}

/*
 * An algorithm returning whether one game formula is a subterm of the other (or
 * if they are equal).
 *
 * Input:  two Game formulae
 * Output: a bool, true if the left Game formula is a subterm of the right, and
 *         false otherwise.
 *
 * One Game formula is a subterm of the other, if the one formula appears in the
 * other formula, which is the same as saying the one Game formula is a descendant
 * of the other.
 */
bool subTerm(Game g, Game h) {
        if (/g := h)
                return true;
        return false;
}
```

Listing 7: ”RuleApplications.rsc”

```
module Proof_Search::RuleApplications

import CloG_Base::GLASTs;
import Proof_Search::Util;

import List;
import Set;

MaybeName canClose(CloGSequent sequent, CloSeqs gfpUnfoldings) {
    for (gfp:term(\mod(dIter(_),_),a,_) ← sequent, x ← a) {
      FpSeq pastGfpU = gfpUnfoldings[x];
      if (pastGfpU.fpFormula.s == gfp.s && pastGfpU.fpFormula.label + x <= a) {
        if (emptyLabels(toList(pastGfpU.contextSeq)) <= emptyLabels(sequent) && (
          true |
          it && (
            false |
            it || past.label <= current.label |
            CloGTerm current ← sequent,
            current.s == past.s
          ) |
          CloGTerm past ← pastGfpU.contextSeq
        )) {
          return closeName(x);
        }
      }
    }
    return noClose();
}

list[MaybeTerm] canApply(ax1(), CloGSequent sequent, _, _) {
    for (CloGTerm term1 ← sequent, atomP(prop(a)) := term1.s) {
        for (CloGTerm term2 ← sequent, neg(atomP(prop(a))) := term2.s) {
            return [appTerms(term1,term2)];
        }
    }
    return [];
}

list[MaybeTerm] canApply(clo(_), CloGSequent sequent, CloSeqs gfpUnfoldings, _) = [
    appTerm(gfp)
    | CloGTerm gfp ← sequent,
      term(\mod(dIter(_),_),_,_) := gfp,
      (
        true
        | it && fpLessThanOrEqualTo(gfp.s, gfpUnfoldings[name].fpFormula.s)
        | CloGName name ← gfp.label
      )
];

list[MaybeTerm] canApply(iterR(), CloGSequent sequent, CloSeqs gfpUnfoldings, set[FpSeq] lfpUnfoldings) {
    for (CloGTerm lfp ← sequent, term(\mod(iter(_),_),_,_) := lfp, <toSet(sequent),lfp> notin lfpUnfoldings
      if ((
        true
        | it && fpLessThanOrEqualTo(lfp.s, gfpUnfoldings[name].fpFormula.s)
        | CloGName name ← lfp.label
      )) {
        return [appTerm(lfp)];
      }
    }
    return [];
}

list[MaybeTerm] canApply(modm(), CloGSequent sequent, _, _) = [
    appTerms(atomMod, atomDMod)
    | CloGTerm atomMod ← sequent,
      term(\mod(atomG(agame(game)),_),_,_) := atomMod,
      CloGTerm atomDMod ← sequent,
      term(\mod(dual(atomG(agame(dgame))),_),_,_) := atomDMod,
      game == dgame
];

list[MaybeTerm] canApply(andR(), CloGSequent sequent, _, _) {for (CloGTerm term ← sequent, and(_,_) := ter
list[MaybeTerm] canApply(orR(), CloGSequent sequent, _, _) {for (CloGTerm term ← sequent, or(_,_) := term.
list[MaybeTerm] canApply(choiceR(), CloGSequent sequent, _, _) {for (CloGTerm term ← sequent, \mod(choice(
list[MaybeTerm] canApply(dChoiceR(), CloGSequent sequent, _, _) {for (CloGTerm term ← sequent, \mod(dChoic
list[MaybeTerm] canApply(concatR(), CloGSequent sequent, _, _) {for (CloGTerm term ← sequent, \mod(concat(
list[MaybeTerm] canApply(testR(), CloGSequent sequent, _, _) {for (CloGTerm term ← sequent, \mod(\test(_),
list[MaybeTerm] canApply(dTestR(), CloGSequent sequent, _, _) {for (CloGTerm term ← sequent, \mod(dTest(_)

CloGProof toAxiom(CloGSequent root, appTerms(term1,term2)) {
  CloGSequent end = [term(term1.s,[],true), term(term2.s,[],true)];
  CloGProof axiom = CloGUnaryInf(end, ax1(), CloGLeaf());

  for (CloGTerm atom ← [term1,term2]) {
    list[CloGName] exps = [];
    for (CloGName name ← atom.label) {
      exps += name;
      axiom = CloGUnaryInf(dup([term(atom.s,exps,true)] + [term[active=false] | CloGTerm term ←
axiom.seq, term.s != atom.s]), exp(), axiom);
    }
  }

  return weakenTo(root, axiom);
}

CloGProof dischargeClosure(CloGSequent root, CloGName name, FpSeq gfpUnfolding) {
  CloGProof closure = disClo(dup([term(gfpUnfolding.fpFormula.s,gfpUnfolding.fpFormula.label + [name],true
                    + [term[active=false] | CloGTerm term ← gfpUnfolding.contextSeq, term != gfpUnfolding.

  list[CloGName] rootNames = head([term(s,label,false) | term(s,label,_) ← root, gfpUnfolding.fpFormula.s
  list[CloGName] names = gfpUnfolding.fpFormula.label + [name];
  list[CloGName] newNames = (rootNames - names);

  for (CloGName newName ← newNames) {
    CloGTerm currentGfp = term(gfpUnfolding.fpFormula.s,names,true);
    names += [newName];
    closure = CloGUnaryInf(
      dup([term(gfpUnfolding.fpFormula.s,names,true)]
      + [term[active=false] | CloGTerm term ← closure.seq, term != currentGfp]),
      exp(),
      closure
    );
  }

  return weakenTo(root, closure);
}

CloGProof weakenTo(CloGSequent from, CloGProof to) {
  CloGProof result = to;
  CloGSequent upper = [upTerm[active=false] | upTerm ← result.seq];
  for (CloGTerm term ← from, term notin upper) {
    CloGSequent upper = [upTerm[active=false] | upTerm ← result.seq];
    result = CloGUnaryInf(dup([term[active=true]] + upper), weak(), result);
  }

  return result;
}

CloGSequent applyMod(appTerms(term(\mod(_,p1),label1,_),term(\mod(_,p2),label2,_))) = [term(p1,label1,false

tuple[CloGSequent left, CloGSequent right] applyConjunct(CloGSequent seq, CloGTerm conjunct) = <
  dup(allExcept(seq, conjunct) + [conjunct[s=conjunct.s.pL]]),
  dup(allExcept(seq, conjunct) + [conjunct[s=conjunct.s.pR]])
>;

CloGSequent apply(orR(), CloGSequent seq, CloGTerm active)
  = dup(allExcept(seq, active)
  + [active[s=active.s.pL], active[s=active.s.pR]]);
CloGSequent apply(choiceR(), CloGSequent seq, CloGTerm active)
  = dup(allExcept(seq, active) +
  [active[s=or(\mod(active.s.g.gL,active.s.pr),\mod(active.s.g.gR,active.s.pr))]]);
CloGSequent apply(dChoiceR(), CloGSequent seq, CloGTerm active)
  = dup(allExcept(seq, active) +
  [active[s=and(\mod(active.s.g.gL,active.s.pr),\mod(active.s.g.gR,active.s.pr))]]);
CloGSequent apply(concatR(), CloGSequent seq, CloGTerm active)
  = dup(allExcept(seq, active) +
  [active[s=\mod(active.s.g.gL,\mod(active.s.g.gR,active.s.pr))]]);
CloGSequent apply(iterR(), CloGSequent seq, CloGTerm active)
  = dup(allExcept(seq, active)
  + [active[s=or(active.s.pr,\mod(active.s.g.ga,\mod(active.s.g,active.s.pr)))]]);
CloGSequent apply(clo(CloGName n), CloGSequent seq, CloGTerm active)
  = dup(allExcept(seq, active) +
  [active[s=and(active.s.pr,\mod(active.s.g.ga,\mod(active.s.g,active.s.pr)))][label=active.label+[n]]]);
CloGSequent apply(testR(), CloGSequent seq, CloGTerm active)
  = dup(allExcept(seq, active) +
  [active[s=and(active.s.g.p,active.s.pr)]]);
```

```
CloGSequent apply(dTestR(), CloGSequent seq, CloGTerm active)
  = dup(allExcept(seq, active) +
  [active[s=or(active.s.g.p,active.s.pr)]]);
CloGSequent apply(CloGRule _, CloGSequent _, CloGTerm _) = [];

CloGSequent allExcept(CloGSequent seq, CloGTerm ex) {
  return [term[active=false] | CloGTerm term ← seq, term != ex];
}
```

Listing 8: "ProofSearch.rsc"

```
module Proof_Search::ProofSearch

import CloG_Base::GLASTs;

import Proof_Search::Util;
import Proof_Search::RuleApplications;

import CloG_Base::LaTeXOutput;

import List;
import Set;
import IO;

MaybeProof findProof(CloGSequent sequent) {
    MaybeProof possibleProof = findProof(sequent, [], (), {}, 0);
    if (proofN(p,n) := possibleProof) {
        // print("<n> nodes visited\n");
        return proof(replaceUnusedClos(p));
    }
    // if (noProof(n) := possibleProof) {
    //     print("<n> nodes visited\n");
    // }
    return possibleProof;
}

MaybeProof findProof(CloGSequent sequent, list[CloGName] branchNames, CloSeqs gfpUnfoldings, set[FpSeq] lfp
    list[MaybeTerm] actives = canApply(ax1(), sequent, gfpUnfoldings, lfpUnfoldings);
    if (actives != [] && appTerms(term1,term2) := head(actives)) {
        return proofN(toAxiom(sequent, appTerms(term1,term2)), n+1);
    }

    MaybeName cloName = canClose(sequent, gfpUnfoldings);
    if (closeName(x) := cloName) {
        return proofN(dischargeClosure(sequent, x, gfpUnfoldings[x]), n+1);
    }

    list[MaybeTerm] gfps = canApply(clo(name("x")), sequent, gfpUnfoldings, lfpUnfoldings);
    for (appTerm(gfp) ← gfps) {
        CloGName newName = getNewName(branchNames);
        CloSeqs newGfpUnfoldings = gfpUnfoldings + (newName:<toSet(sequent),gfp>);
        CloGSequent nextSequent = apply(clo(newName), sequent, gfp);
        list[CloGName] newNames = branchNames + [newName];

        MaybeProof maybeProof = findProof(nextSequent, newNames, newGfpUnfoldings, lfpUnfoldings, n);

        if (proofN(p,nodes) := maybeProof) {
            return proofN(CloGUnaryInf([term[active=(term==gfp)] | term ← sequent], clo(newName), p), nodes
        }
        n = maybeProof.n;
    }

    for (CloGRule rule ← [orR(), choiceR(), concatR(), testR(), dTestR(), dChoiceR(), iterR()]) {
        list[MaybeTerm] actives = canApply(rule, sequent, gfpUnfoldings, lfpUnfoldings);
        if (actives != [] && appTerm(active) := head(actives)) {
            CloGSequent nextSequent = apply(rule, sequent, active);

            set[FpSeq] newLfpUnfoldings = lfpUnfoldings;
            if (rule == iterR()) {
                newLfpUnfoldings = lfpUnfoldings + {<toSet(sequent),active>};
            }

            MaybeProof maybeProof = findProof(nextSequent, branchNames, gfpUnfoldings, newLfpUnfoldings, n

            if (proofN(p,nodes) := maybeProof) {
                return proofN(CloGUnaryInf([term[active=(term==active)] | term ← sequent], rule, p),nodes+
            }
            return noProof(maybeProof.n + 1);
        }
    }

    for (CloGRule rule ← [andR()]) {
        list[MaybeTerm] conjuncts = canApply(rule, sequent, gfpUnfoldings, lfpUnfoldings);
        if (conjuncts != [] && appTerm(conjunct) := head(conjuncts)) {
            tuple[CloGSequent left, CloGSequent right] nextSequents = applyConjunct(sequent, conjunct);

            int nodesLeft = 0;
            int nodesRight = 0;
            MaybeProof maybeLeft = findProof(nextSequents.left, branchNames, gfpUnfoldings, lfpUnfoldings,
            if (proofN(pL, nodes1) := maybeLeft) {
                nodesLeft = nodes1;
                MaybeProof maybeRight = findProof(nextSequents.right, branchNames, gfpUnfoldings, lfpUnfol
                if (proofN(pR, nodes2) := maybeRight) {
                    return proofN(CloGBinaryInf([term[active=(term==conjunct)] | term ← sequent], pL, pR),
                }
                nodesRight = maybeRight.n;
            } else {
                nodesLeft = maybeLeft.n;
            }
            n = nodesLeft + nodesRight;
        }
    }

    for (appTerms(mod1,mod2) ← canApply(modm(), sequent, gfpUnfoldings, lfpUnfoldings)) {
        CloGSequent nextSequent = applyMod(appTerms(mod1,mod2));
        MaybeProof maybeProof = findProof(nextSequent, branchNames, gfpUnfoldings, lfpUnfoldings, n);

        if (proofN(p,nodes) := maybeProof) {
            CloGProof modInf = CloGUnaryInf([mod1[active=true], mod2[active=true]], modm(), p);

            return proofN(weakenTo(sequent, modInf), nodes+1);
        }
        n = maybeProof.n;
    }

    return noProof(n+1);
}
```

A.3 Package “Bound_Calculation”

Listing 9: ”SequentClosure.rsc”

```
module Bound_Calculation::SequentClosure

import CloG_Base::GLASTs;
import CloG_Base::LaTeXOutput;

import util::Math;

import Set;
import List;

int closureCombinations(CloGSequent root) = round(pow(2,size(getClosure(root))))-1;

CloGSequent getClosure(CloGSequent root) {
    return getClosure(root, {});
```

```
}

CloGSequent getClosure(CloGSequent seq, CloGSequent closed) {
    newSeq = seq;
    newClosed = closed;

    notClosed = toList(seq - closed);

    if (size(notClosed) == 0) {
        return seq;
    }

    for (CloGTerm termToClose ← notClosed) {
        next = getNextTerms(termToClose);

        //print("$$" + LaTeXCloGTree(termToClose) + "\\Rightarrow" + LaTeXCloGSequent(toList(next), false)

        newSeq += next;
        newClosed += {termToClose};
    }

    return getClosure(newSeq, newClosed);
}

private CloGSequent getNextTerms(term(and(pL,pR),label,_)) = {term(pL,label,false),term(pR,label,false)};
private CloGSequent getNextTerms(term(or(pL,pR),label,_)) = {term(pL,label,false),term(pR,label,false)};
private CloGSequent getNextTerms(term(\mod(\test(p),pr),label,_)) = {term(and(p,pr),label,false)};
private CloGSequent getNextTerms(term(\mod(dTest(p),pr),label,_)) = {term(or(p,pr),label,false)};
private CloGSequent getNextTerms(term(\mod(iter(ga),pr),label,_)) = {term(or(pr,\mod(ga,\mod(iter(ga),pr)))
private CloGSequent getNextTerms(term(\mod(dIter(ga),pr),label,_)) = {term(and(pr,\mod(ga,\mod(dIter(ga),pr
private CloGSequent getNextTerms(term(\mod(concat(gaL,gaR),pr),label,_)) = {term(\mod(gaL,\mod(gaR,pr)),lab
private CloGSequent getNextTerms(term(\mod(dChoice(gaL,gaR),pr),label,_)) = {term(and(\mod(gaL,pr),\mod(gaR
private CloGSequent getNextTerms(term(\mod(choice(gaL,gaR),pr),label,_)) = {term(or(\mod(gaL,pr),\mod(gaR,p
private CloGSequent getNextTerms(term(\mod(g,pr),label,_)) = {term(pr,label,false)};
private CloGSequent getNextTerms(CloGTerm _) = {};
```

Listing 10: ”CombinationCalculation.rsc”

```
module Bound_Calculation::CombinationCalculation

import CloG_Base::GLASTs;
import CloG_Base::LaTeXOutput;

import util::Math;

import IO;
import Set;
import List;

int getCombinations(CloGSequent root) {
    return getCombinations(root, {}, {});
}

int getCombinations(CloGSequent seq, CloGSequent closed, CloGSequent ands) {
    notClosed = toList((seq - ands) - closed);

    if (size(notClosed) == 0) {
        foundAnds = [andTerm | CloGTerm andTerm ← (seq - ands), term(and(_,_),_,_) := andTerm];

        if (size(foundAnds) > 0 && term(and(pL,pR),label,active) := head(foundAnds)) {
            newAnds = ands + {term(and(pL,pR),label,active)};
            return min(
                getCombinations(seq + {term(pL,label,false)}, closed, newAnds),
                getCombinations(seq + {term(pR,label,false)}, closed, newAnds)
            );
        }

        foundMod = [modTerm | CloGTerm modTerm ← (seq - ands), term(and(_,_),_,_) := modTerm];
        print("$$<LaTeXCloGSequent(toList(seq),false)>$$\n");
        return round(pow(2, size(seq)))-1;
    }

    newSeq = seq;
    newClosed = closed;

    for (CloGTerm termToClose ← notClosed) {
        next = getNextTerms(termToClose);

        newSeq += next;
        newClosed += {termToClose};
    }

    return getCombinations(newSeq, newClosed, ands);
}

private CloGSequent getNextTerms(term(or(pL,pR),label,_)) = {term(pL,label,false),term(pR,label,false)};
private CloGSequent getNextTerms(term(\mod(\test(p),pr),label,_)) = {term(and(p,pr),label,false)};
private CloGSequent getNextTerms(term(\mod(dTest(p),pr),label,_)) = {term(or(p,pr),label,false)};
private CloGSequent getNextTerms(term(\mod(iter(ga),pr),label,_)) = {term(or(pr,\mod(ga,\mod(iter(ga),pr)))
private CloGSequent getNextTerms(term(\mod(dIter(ga),pr),label,_)) = {term(and(pr,\mod(ga,\mod(dIter(ga),pr
private CloGSequent getNextTerms(term(\mod(concat(gaL,gaR),pr),label,_)) = {term(\mod(gaL,\mod(gaR,pr)),lab
private CloGSequent getNextTerms(term(\mod(dChoice(gaL,gaR),pr),label,_)) = {term(and(\mod(gaL,pr),\mod(ga
private CloGSequent getNextTerms(term(\mod(choice(gaL,gaR),pr),label,_)) = {term(or(\mod(gaL,pr),\mod(gaR,
private CloGSequent getNextTerms(CloGTerm _) = {};
```