intuitR: A Theorem Prover for Intuitionistic Propositional Logic

Erik Rauer
rauer007@morris.umn.edu
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA

Abstract

A constructive proof proves the existence of a mathematical object by giving the steps necessary to construct said object. Proofs of this type can be interpreted as an algorithm for creating such an object. *Intuitionistic Propositional Logic (IPL)* is a propositional logic system wherein all valid proofs are constructive. *intuitR* is a theorem prover for IPL, that is, it determines whether a given formula is valid in IPL or not. In this paper, we describe how *intuitR* determines the validity of a formula and review its performance. When compared on a benchmark set of problems, *intuitR* was determined to solve more problems and to be of comparable speed or better than other IPL-provers.

Keywords: theorem provers, intuitionistic logic, automated deduction, SAT solvers

1 Introduction

Imagine your task is to write a program which takes a planar graph (i.e. a graph whose edges do not overlap) as the input and which outputs the same graph with each vertex assigned a color such that no two adjacent vertices have the same color. You want this coloring to use as few colors as possible and you want to be confident that the output satisfies the criteria given. After doing some research you come across the Four Color Theorem, which states that at most four colors are necessary to color a planar graph as described. While reading the proof of this theorem, you realize that it gives a step-by-step process with which you can properly color any planar graph using only four colors. So, you write your program based on this process.

A proof such as the one described above, where the existence of some mathematical object is proved by explicitly showing how to construct said object, is called a *constructive proof*. As implied in the Four Color Theorem example, a constructive proof of a property can be interpreted as an algorithm for creating an object with said property. While it might not necessarily be the most efficient, such an algorithm has been mathematically shown to provide the desired output. This equivalence between constructive proofs and algorithms was formalised as the Curry-Howard correspondence [6].

As an alternate example, consider trying to prove the existence of irrational numbers a and b, such that a^b is rational. A non-constructive proof might be as follows: Assume that $\sqrt{2}^{\sqrt{2}}$ is either rational or irrational. If it is rational, let $a = \sqrt{2}$ and $b = \sqrt{2}$ and the proof is complete. If it is irrational, let $a = \sqrt{2}^{\sqrt{2}}$ and let $b = \sqrt{2}$. Then $a^b = \left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^2 = 2$, which is rational, so the proof is complete. This proof is non-constructive since we never actually show whether $\sqrt{2}^{\sqrt{2}}$ is rational or irrational, we just show that no matter which of these two situations is the case, a pair of irrational numbers that fits the given criteria can be found.

Intuitionistic Logic is a system of logic in which all valid proofs are constructive. Intuitionistic logic is the same as classical logic, except that it does not allow the Law of Excluded Middle $p \vee \neg p$, or the Law of Double Negation Elimination $\neg\neg p \equiv p$ [7]. Notice that in the non-constructive example above, the first step is to apply $p \vee \neg p$, where $p = \left(\sqrt{2}^{\sqrt{2}} \textit{ is rational}\right)$, making the proof invalid in intuitionistic logic. *Intuitionistic Propositional Logic (IPL)* is the propositional form of intuitionisitic logic. IPL uses the same logical symbols as classical propositional logic, namely conjunction $\wedge$, disjunction $\vee$, implication $\rightarrow$, true $\top$, and false $\bot$. We also include negation $\neg$, where $\neg P$ is considered shorthand for $P \rightarrow \bot$. Note that IPL is a propositional logic, so it does not include the quantifiers $\forall$ and $\exists$.

For a given logical formula α, we say that α is *provable* or *valid* in a system of logic, if the axioms of said system can be used to derive α. If α can not be derived from these axioms, it is considered *unprovable* or *not valid*. A *theorem prover* is a program that, when given a formula α, determines whether it is valid or not. *intuitR* is a theorem prover for Intuitionistic Propositional Logic, which was created by Fiorentini [4]. It is based on another theorem prover, *intuit*, which was created by Claessen and Rosén [2]. Both *intuit* and *intuitR* take a given propositional formula α as input and return whether or not α is valid in intuitionistic propositional logic (*IPL-valid*) or not. Theorem provers of this type are also called *IPL-provers*.

In this paper, we discuss *intuitR* and how it works in detail. To start, Section 2 will introduce SAT solvers (2.1) and

Kripke Semantics (2.2), two necessary background concepts that *intuitR* heavily relies upon. Section 3 explains the *intuitR* prover itself. This includes the clausification procedure which converts the formula into a special form for *intuitR* to use, the two key logic rules *intuitR* is based on, and the *proveR* algorithm itself, which determines the validity of the given formula. Finally, *intuitR*'s speed and ability to correctly solve IPL-validity problems is compared to other IPL provers, including *intuit*. The methods and results of this comparison are detailed in Section 4.

2 Background

In this section, we introduce the important background concepts needed to understand *intuitR*. This includes SAT solvers, a tool used to determine the validity of formulas in classical logic, and Kripke semantics, which are a formal way to determine whether a formula is IPL-valid or not.

2.1 SAT Solvers

The key factor that differentiates *intuit* and *intuitR* from other IPL-provers is the use of satisfiability (SAT) solvers.

A *satisfiability solver* is an algorithm which solves the *Boolean Satisfiability Problem (SAT)*: Given a propositional formula α, is there an assignment of the propositional variables in α to either $TRUE$ or $FALSE$, such that α is satisfiable (i.e. evaluates to $TRUE$)? If yes, the SAT solver will return said assignment [5]. For example, if given the formula $\alpha = p \wedge \neg q$, a SAT solver would return that α is satisfiable with $p = TRUE$ and $q = FALSE$.

Both *intuit* and *intuitR* use an implementation of the *MINISAT* SAT solver [3]. However, both provers can be run using any SAT solver, so long as it supports the following operations [4]:

newSolver()

- Create a new SAT solver

addClause(s, ρ)

- Add clause (a disjunction of propositional variables) ρ to SAT solver's existing clauses $R(s)$

satProve(s, A, g)

- Use SAT solver s to prove g based on the clauses $R(s)$ that have already been added and the set of propositional variables A that are assumed to be true
- Returns:
  - *YES(A')*, if $R(s)$ and $A' \subseteq A$ being true makes g true
  - *NO(M)*, if $R(s)$ and the set of propositional variables $M \supseteq A$ are both true, but g is false

2.2 Kripke Semantics and Models

There are a number of different ways to determine if a given propositional formula is IPL-valid, such as Beth's tableaux, Heyting Algebras, and Kripke semantics [7]. Of these, both

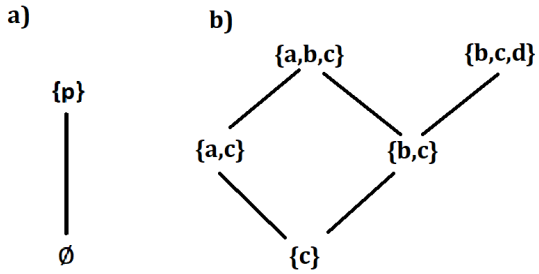

Figure 1. Visualization of two Kripke models. a) is a counter model for $p \vee \neg p$ as described in Section 2.2. b) is an arbitrary example Kripke model. Bottom nodes ($\emptyset$ in a and $\{c\}$ in b) are the roots of the models.

intuit and *intuitR* rely on Kripke semantics, which work as follows:

For any propositional formula α, there exist a finite number of Kripke models, which will be described in detail below. A Kripke model κ can either force the propositional formula, denoted $\kappa \vDash \alpha$, or not force the formula, denoted $\kappa \nvDash \alpha$, which will be defined below. The formula α is IPL-valid iff for every possible Kripke model κ, $\kappa \vDash \alpha$. A Kripke model where $\kappa \nvDash \alpha$ is called a Kripke counter model for α. So determining the IPL-validity of α is the same as determining whether or not a Kripke counter model of α can be found.

A Kripke model is a quadruple $(W, \leq, r, \delta)$, where W is a set of nodes, or worlds, and δ is a mapping from each world to a set of propositional variables that are considered true in that world. Additonally, $\leq$ is a partial ordering of the worlds in W such that for all $k, k' \in W$, if $k \leq k'$, then $\delta(k) \subseteq \delta(k')$. The world $r \in W$ is called the root and is the minimum world, i.e. $r \leq k$ for every other world $k \in W$ [7]. Kripke models can be thought of as a tree-like graph structure, where each world is a node in the tree containing the set of propositional variables associated with that world. Then r is the root of the tree and each branch is a chain of worlds that are $\leq$ to each other (see Figure 1). Note that this is not an actual tree since the branches can merge back together.

For a Kripke model with a set of worlds W, every world $k \in W$ can either force a propositonal formula α, denoted by $k \vDash \alpha$, or not force the formula, denoted by $k \nvDash \alpha$. A world k forces a propositional formula based on the following rules [7]:

$$k \vDash p, \text{ if } p \in \delta(k) \tag{1}$$

$$k \nvDash \bot \tag{2}$$

$$k \vDash P \wedge Q, \text{ if } k \vDash P \text{ and } k \vDash Q \tag{3}$$

$$k \vDash P \vee Q, \text{ if } k \vDash P \text{ or } k \vDash Q \tag{4}$$

$$k \vDash \neg P, \text{ if for all } k' \geq k, k' \nvDash P \tag{5}$$

$$k \vDash P \rightarrow Q, \text{ if for every } k' \geq k, \text{ if } k' \vDash P, \text{ then } k' \vDash Q \tag{6}$$

In IPL, a Kripke model with root r forces a propositional formula α iff $r \vDash \alpha$ [7].

As an example, consider the formula $\alpha = p \vee \neg p$, which we know from the definition of intuitionistic logic to not be IPL-valid. Now consider the Kripke model $\kappa = (\{w, w'\}, \leq, w, \delta)$, where $\delta(w) = \emptyset$, $\delta(w') = \{p\}$, and $w < w'$ (see Figure 1) [7]. Since w is the root, in order for $\kappa \vDash \alpha$, it must be true that $w \vDash \alpha$. So from (4) above, in order for this to be the case either $w \vDash p$ or $w \vDash \neg p$. Since $w = \emptyset$, $w \nvDash p$ by (1). Additionally, from (5) above, $w \vDash \neg p$ only if both $w \nvDash p$ and $w' \nvDash p$. But since $\delta(w') = \{p\}$, (1) says that $w' \vDash p$, making $w \nvDash \neg p$. So $w \nvDash \alpha$, making κ a counter model for α.

3 *intuitR*

This section focuses on how the *intuitR* theorem prover works. To start, we describe the clausification procedure which converts a formula into a special form for *intuitR* to utilize. Next, the two logic rules *intuitR* is based upon are described and we show how they can be used to determine the IPL-validity of a formula. Finally, the *proveR* algorithm, which determines the IPL-validity of a given formula, is explained.

3.1 Clausification

In order to work with a given propositional formula α, both *intuit* and *intuitR* require the formula to be in a special form. This special form has the same IPL-validity as α, it is just easier for *intuit* and *intuitR* to work with. This is done through the *clausification* procedure detailed in [2] and which will be described below. The final form of this procedure consists of a set of *flat clauses* and a set of *implication clauses*. A flat clause is a formula of the shape:

$$(a_1 \wedge a_2 \wedge \ldots \wedge a_n) \rightarrow (b_1 \vee b_2 \vee \ldots \vee b_m)$$

where $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$ are propositional variables. Note that when $n = 0$, a flat clause simplifies to the form $b_1 \vee b_2 \vee \ldots \vee b_m$. An implication clause on the other hand is a formula of the shape:

$$(a \rightarrow b) \rightarrow c$$

where a, b, c are all propositional variables. Through the clausification procedure, a propositional formula α gets converted into the form

$$\left(\bigwedge R \wedge \bigwedge X\right) \rightarrow g \tag{7}$$

where g is a propositional variable and $\bigwedge$ denotes set conjunction, i.e. $\bigwedge A = (a_1 \wedge a_2 \wedge \ldots \wedge a_n)$ for $A = \{a_1, a_2, \ldots, a_n\}$. Additionally, in this case and for the rest of the paper, R will denote a set of flat clauses and X will denote a set of implication clauses.

The first step in the clausification of the formula α is to turn it into the form $A \rightarrow g$ where g is a propositional variable. If it is not already in this form, we simply introduce a new variable g and change it to $(\alpha \rightarrow g) \rightarrow g$, which has the same IPL-validity as α [2]. From there a combination of the transformations in Figure 2 are applied to the subformula A from $A \rightarrow g$ until the formula is in the form shown in Equation 7. Transformations (1) and (2) are used to introduce implications to sub-formulas, so that they may be used by the other transformations. Transformations (3), (4), and (5) ensure that the formula has the desired shape of primarily flat and implication clauses. The final five transformations "pull" sub-formulas out of the existing flat and implication clauses by introducing new propositional variables. Finally, we require that for each implication clause $(a \rightarrow b) \rightarrow c \in X$, the flat clause $(b \rightarrow c)$ is in R, where R and X are the sets in Equation 7. This can be done by simply taking every $(a \rightarrow b) \rightarrow c \in X$ and adding $(b \rightarrow c)$ to R as the final step in the clausification procedure [2]. We call the resulting formula a *reduced-sequent*, or *r-sequent* for short, and denote it as $R, X \Rightarrow g$.

$$\begin{aligned}
A &\twoheadrightarrow \top \rightarrow A && (1)\\
\neg A &\twoheadrightarrow A \rightarrow \bot && (2)\\
(A \vee B) \rightarrow p &\twoheadrightarrow (A \rightarrow p) \wedge (B \rightarrow p) && (3)\\
p \rightarrow (A \wedge B) &\twoheadrightarrow (p \rightarrow A) \wedge (p \rightarrow B) && (4)\\
A \rightarrow (B \rightarrow C) &\twoheadrightarrow (A \wedge B) \rightarrow C && (5)\\
A \rightarrow (\ldots \vee B \vee \ldots) &\twoheadrightarrow (A \rightarrow (\ldots \vee b \vee \ldots)) \wedge (b \rightarrow B) && (6)\\
(\ldots \wedge A \wedge \ldots) \rightarrow B &\twoheadrightarrow ((\ldots \wedge a \wedge \ldots) \rightarrow B) \wedge (A \rightarrow a) && (7)\\
(A \rightarrow B) \rightarrow C &\twoheadrightarrow ((a \rightarrow B) \rightarrow C) \wedge (a \rightarrow A) && (8)\\
(A \rightarrow B) \rightarrow C &\twoheadrightarrow ((A \rightarrow b) \rightarrow C) \wedge (B \rightarrow b) && (9)\\
(A \rightarrow B) \rightarrow C &\twoheadrightarrow ((A \rightarrow B) \rightarrow c) \wedge (c \rightarrow C) && (10)
\end{aligned}$$

Figure 2. Rules used for the clausification procedure. A, B, C are all subformulas, p is a propositional variable, and a, b, c are newly introduced propositional variables. From [2].

As an example, consider applying the clausification procedure to the propositional formula $\alpha = p \vee \neg p$. Since α is not in the form $A \rightarrow g$, we first introduce a new variable g as described above, resulting in the formula $((p \vee \neg p) \rightarrow g) \rightarrow g$. Next, the rules of Figure 2 are applied to the LHS of the formula as follows:

$$\begin{aligned}
&((p \vee \neg p) \rightarrow g) \rightarrow g\\
\twoheadrightarrow &((p \rightarrow g) \wedge (\neg p \rightarrow g)) \rightarrow g && \text{(Transf. 3)}\\
\twoheadrightarrow &((p \rightarrow g) \wedge ((p \rightarrow \bot) \rightarrow g)) \rightarrow g && \text{(Transf. 2)}
\end{aligned}$$

This is in the same shape as in Equation 7, with $R = \{p \rightarrow g\}$ and $X = \{(p \rightarrow \bot) \rightarrow g\}$. Finally, we add $\bot \rightarrow g$ to R, resulting in the r-sequent with $R = \{p \rightarrow g, \bot \rightarrow g\}$ and $X = \{(p \rightarrow \bot) \rightarrow g\}$.

Similar to other propositional formulas, an r-sequent can be considered IPL-valid or not IPL-valid. We say an r-sequent $R, X \Rightarrow g$ is IPL-valid if for every Kripke model (see Section 2.2) κ with root r, $r \vDash (R \cup X)$ implies that $r \vDash g$. So if there exists any Kripke model κ where $r \vDash (R \cup X)$ and $r \nvDash g$, $R, X \Rightarrow g$ is not IPL-valid and we call κ a counter model.

$$cpl_0: \quad \frac{R \vdash_c g}{R, X \Rightarrow g}$$

$$cpl_1: \quad \frac{R, A \vdash_c b \qquad R, \varphi, X \Rightarrow g}{R, X \Rightarrow g} \qquad \begin{array}{r}(a \rightarrow b) \rightarrow c \in X \\ A \subseteq V \\ \varphi = \bigwedge (A \backslash \{a\}) \rightarrow c\end{array}$$

Figure 3. The rules cpl_0 and cpl_1 from [4]. Note that $R \vdash_c g$ and $R, A \vdash_c b$ is notation saying that R proves g in classical logic and that R and A together prove b in classical logic.

By performing the clausification procedure on α we do not change its IPL-validity, i.e. α is IPL-valid iff the associated r-sequent $R, X \Rightarrow g$ produced by the clausification procedure of α is also IPL-valid. So the issue of determining whether a formula is IPL-valid or not can be reduced to determining whether its associated r-sequent is IPL-valid or not [4].

3.2 Logic Rules

intuitR relies on two rules which Fiorentini [4] names cpl_0 and cpl_1 (see Figure 3). cpl_0 is based on the work in [1] and says that if a set of flat clauses R can be used to prove a propositional variable g in classical logic, then the same set of flat clauses can also be used to prove g in IPL. If this is the case, then for any Kripke model κ, such that $\kappa \vDash R$, it must also be true that $\kappa \vDash g$. Thus, any r-sequent with R as the set of flat clauses must be IPL-valid. cpl_0 is one way in which *intuitR* utilizes SAT solvers. By adding all the flat clauses in R to a SAT solver s and then calling *satProve(s, ∅, g)* (see Section 2.1), a SAT solver can be used to determine whether the initial condition of cpl_0 holds. If it does, we can apply cpl_0 to show that an r-sequent with R as the set of flat clauses is IPL-valid.

cpl_1, which is discussed in detail in [4], is best understood "backwards" since that is how it is used by *intuitR*. Given an r-sequent $R, X \Rightarrow g$, cpl_1 says that if a certain condition holds, a new flat clause can be added to R. Doing so gives a new r-sequent $R', X \Rightarrow g$. The IPL-validity of this new r-sequent implies the IPL-validity of the original, i.e. if $R', X \Rightarrow g$ is IPL-valid, then $R, X \Rightarrow g$ must also be IPL-valid. Using cpl_1 requires an implication clause $(a \rightarrow b) \rightarrow c$ from X. For any such $(a \rightarrow b) \rightarrow c \in X$, if the original set of flat clauses R and a set of propositional variables (denoted A) can together be used to prove b in classical logic, then a new flat clause φ can be added to R to create R'. This new flat clause is $\varphi = \bigwedge(A \backslash \{a\}) \rightarrow c$.

Similar to cpl_0, determining whether φ can be added is done by a SAT solver. Namely, for a SAT solver s, we add all the flat clauses from R to s, before calling *satSolve(s, A, b)*. If the SAT solver says that the condition holds, cpl_1 can be applied to add the new φ to R. The problem then becomes finding an appropriate implication clause and the corresponding set of propositional variables A, so that cpl_1 can be used. How *intuitR* decides which clause and set of variables to use is described in Section 3.3 below.

Given an r-sequent $R_0, X \Rightarrow g$, the rules cpl_0 and cpl_1 can be used to determine the IPL-validity of $R_0, X \Rightarrow g$. First, we determine whether R_0 proves g classically. If it does, cpl_0 says that $R_0, X \Rightarrow g$ is IPL-valid. If it does not, it is not necessarily the case that $R_0, X \Rightarrow g$ is not IPL-valid, so we attempt to use cpl_1 "backwards" as described above. To do so, we search for an implication clause that satisfies the condition for cpl_1 to be used and add the corresponding flat clause. In doing so, we have created a new r-sequent $R_1, X \Rightarrow g$. Once again, we check whether cpl_0 can be used to determine the IPL-validity of this new r-sequent. If it can, by cpl_1, the original r-sequent $R_0, X \Rightarrow g$ must also be IPL-valid. If not, we attempt to use cpl_1 again to create another r-sequent $R_2, X \Rightarrow g$. This process of checking cpl_0 and then adding a flat clause via cpl_1 continues until either cpl_0 shows an r-sequent $R_n, X \Rightarrow g$ is IPL-valid, in which case the original r-sequent $R_0, X \Rightarrow g$ is also IPL-valid, or until cpl_1 can not be used to add a new flat clause, in which case the original r-sequent $R_0, X \Rightarrow g$ is not IPL-valid. One of these must eventually happen, since there are a finite number of implication clauses in X, so cpl_1 can only be used a finite number of times until it ends up only being able to add flat clauses already in R_n.

3.3 *proveR* Algorithm

The *proveR* algorithm utilizes the rules from Section 3.2 to determine whether the input formula is IPL-valid or not. As an input, *proveR* takes an r-sequent $R, X \Rightarrow g$ and outputs *Valid* if the r-sequent is IPL-valid, or *CountSat* if it is not.

The algorithm has a nested loop structure, consisting of an "outer loop" (Steps S1 through S6 in Figure 4) and an "inner loop" (Steps S3-S4-S5 in Figure 4). The outer loop implements the process of continuously checking if cpl_0 applies and if it does not, adding a new flat clause via cpl_1 as described in Section 3.2. The inner loop on the other hand, determines which new flat clause should be added via cpl_1. It does so by attempting to create a Kripke counter model for the current r-sequent (which is also a counter model for the input r-sequent) [4].

In order to efficiently create an appropriate Kripke counter model, Fiorentini defines a relation between a world M from a set of worlds W and an implication clause $(a \rightarrow b) \rightarrow c$. This relation is denoted $M \rhd_W (a \rightarrow b) \rightarrow c$ and is true when [4]:

$$(M \vDash a) \text{ or } (M \vDash b) \text{ or } (M \vDash c) \text{ or}$$
$$(\exists M' \in W \text{ such that } M < M' \text{ and } M' \vDash a \text{ but } M' \nvDash b)$$

We say $M \rhd_W X$ if $M \rhd_W \lambda$ for all $\lambda \in X$. This relation is used in *Proposition 2* from [4], which is key in the construction of the Kripke counter model and goes as follows: Let $\sigma = R, X \Rightarrow g$ be an r-sequent and let $\kappa = (W, \leq, M_0, \delta)$ be a Kripke model.

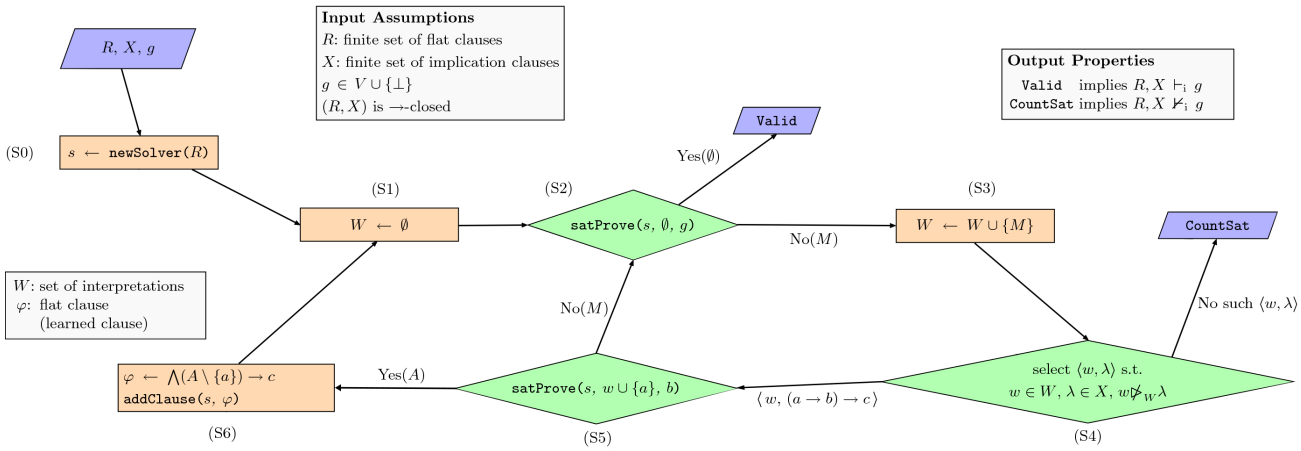

Figure 4. Flow chart of the *proveR* algorithm. Taken from [4]. Note that $R, X \vdash_i g$ is notation saying that the r-sequent $R, X \Rightarrow g$ is IPL-valid. Similarly, $R, X \nvdash_i g$ means that $R, X \Rightarrow g$ is not IPL-valid.

Then, $\kappa \nvDash \sigma$ (i.e. κ is a counter model for σ) iff $g \notin M_0$ and for every $M \in W$, $M \vDash R$ and $M \rhd_W X$. In this context, the $\rhd_W$ relation is an easy way to check whether a given world forces an implication cause.

The *proveR* algorithm, illustrated in Figure 4, works as follows: Given an r-sequent $R, X \Rightarrow g$, we first construct a new SAT solver s and pass it all the flat clauses of R (step $S0$). We then start the outer loop by setting W to $\emptyset$ (step $S1$). Next, we call *satProve(s,∅,g)* (step $S2$), to check whether just the existing flat clauses already in R are enough to prove g classically. If they are, by cpl_0 they are enough to prove g in IPL, making the r-sequent $R, X \Rightarrow g$ IPL-valid, so the algorithm is complete and returns *Valid*. If not, we get a set of propositional variables M, which make the clauses in R true, but make g false. This set M is then added to W (step $S3$).

By adding this M to W, we have started the inner loop. In step $S4$, we attempt to find a set w from W and a corresponding implication clause $\lambda = (a \rightarrow b) \rightarrow c$, such that $w \not\Vdash_W \lambda$. Note that since W consists of sets of propositional variables, its elements can be interpreted as worlds in a Kripke model (see below), allowing us to use the $\rhd$ relation. If no such pair can be found, we have a valid counter model and can return *CounterSat*. If we do find a pair that fits the desired criterion, we call *satProve(s,w ∪ {a},b)* (step $S5$). In this case, if the call to the SAT solver returns *No*, we are given a new set of propositional variables M that make the clauses in R true, but make b false. This set M is also added to W, restarting the inner loop. On the other hand, if we do find a set of propositional variables A, we have met the conditions necessary to apply cpl_1. So, using $\varphi = \bigwedge(A \setminus \{a\}) \rightarrow c$, we can use cpl_1 as described in Section 3.2 to add a new flat clause to R, giving a new r-sequent (step $S6$). From there, we restart the outer loop by resetting W back to the empty set and checking whether cpl_0 can be applied to the new r-sequent.

When *proveR* returns *CounterSat*, a Kripke counter model can fairly easily be constructed from the set W. We create a world k_n for each element $w \in W$ and set the associated $\delta(k_n) = w$. Then $k_m \leq k_l$ if $\delta(k_m) \subseteq \delta(k_k)$. So, the first element to be added to W, call it M_0, is the root of our counter model, since by the definition of *satProve*() (described in Section 2.1) it must be a subset of any M returned in step $S5$. From step $S2$, we know that this root M_0 does not contain g, otherwise g would be true and the SAT solver would not have returned *No*. This also means that M_0 must make all of the flat clauses in R true. Since every other element added to W is a superset of M_0, they must also make all of the flat clauses in R true. Thus every world makes the flat clauses in R true, so $M_0 \vDash R$. *CounterSat* is only returned when no $\lambda \in X$ and no $w \in W$ can be found such that $w \not\Vdash_W \lambda$, or in other words, it is returned when for every $w \in W$ and every $\lambda \in X$, $w \rhd_W \lambda$. So *Proposition 2* from above applies to the model we have constructed, making it a valid counter model to the current r-sequent and thus the original r-sequent [4]. Note that the inner loop must eventually terminate, since for every call in step $S5$ that returns the set M, $a \in M$ and $b \notin M$. By adding M to W, the pair of w and λ selected in that iteration can not be selected again, since doing so makes $w \rhd_W \lambda$ by the fourth condition of the $\rhd_W$ relation.

4 Experiment and Results

In order to compare *intuitR* and *intuit* with other IPL-provers, they were both implemented in Haskell, using the *MINISAT* [3] SAT solver. In addition to *intuit*, *intuitR* was compared to the IPL-provers *fCube* and *intHistGC* [4]. These provers were run on a set of sample benchmark problems, which *inuit* was intially tested with and are described in detail in [2]. This set consists of a total of 1200 benchmark problems, sorted into 32 groups. 498 of these problems are

Problem Set (Number of Problems)	*intuitR*	*intuit*	*fCube*	*intHistGC*
SYJ201 (50)	**50 (2.259)**	50 (11.494)	50 (259.776)	50 (39.466)
SYJ207 (50)	**50 (2.291)**	50 (109.919)	50 (138.546)	50 (1014.476)
SYJ211 (50)	**50 (0.462)**	50 (1.251)	50 (1.073)	50 (63.686)
SYJ212 (50)	**50 (0.669)**	42 (587.794)	50 (2.698)	50 (1.624)
EC (100)	100 (2.738)	100 (0.821)	100 (6.183)	**100 (0.651)**
negEC (100)	100 (3.614)	**100 (1.116)**	100 (13.733)	100 (5.807)
portia (100)	100 (32.878)	**100 (22.596)**	100 (3255.818)	100 (3200.135)
Total Unsolved	**28**	36	43	38
Total Time (For These Problems)	**44.911**	734.991	3677.827	4325.845

Table 1. Comparison of different IPL provers on selected benchmark problem sets. Each cell contains the number of problems each prover could solve, followed by amount of time it took to solve said problems. The fastest prover for each problem set is in bold. Only the most relevant data from [4] is shown.

formulas that are IPL-valid and the other 702 of them are not IPL-valid. The provers were all run on the same computer and given a timeout of 600 seconds per problem. Both the number of problems each prover could solve and the amount of time it took to solve each problem were recorded.

The most relevant results of the comparison between the four provers is shown in Table 1. As can be seen, *intuitR* solved all but 28 of the benchmark problems within the 600 second timeout, which is 8 more than the next best prover *intuit* was able to solve. For the benchmark problem sets SYJ201, SYJ207, SYJ211, and SYJ212, *intuitR* was significantly faster than the next best IPL-prover. Additionally, the only problem sets where *intuitR* was significantly slower than another solver are EC, negEC, and portia. Otherwise, *intuitR* was of comparable speed to the best solver (which was usually *intuit*) for all of the remaining benchmark problem sets not shown in Table 1 [4]. Despite being slower on three problem sets, *intuitR* took less total time to solve all of the benchmark problems than any of the other provers.

Fiorentini believes that *intuitR* is generally faster than these other provers because *intuitR* only relies on two relatively simple logical rules and because the counter models *intuitR* generates are typically smaller than those of the other provers. Unlike *intuitR* which only relies on two rules, *intuit* relies on three, much more general rules than cpl_0 and cpl_1 [4]. This results in much more complicated derivations containing lots of branches, as opposed to the more linear structure that comes from applying cpl_0 and cpl_1 as described in Section 3.2. Finding these more complicated derivations tends to require more calls to a SAT solver, which slows the theorem prover down. Additionally, every iteration of the outer loop of *proveR* "resets" the counter model that is being built by the inner loop, which results in *intuitR* usually generating smaller counter models. Smaller counter models means there are less worlds to check for forcing, which also speeds the theorem prover up. Problem 2 from SYJ207 is given as an example. For this problem, *intuit* called the SAT solver 31 times and created a counter model containing 6 worlds, while the counter model generated by *intuitR* only required 4 worlds and 14 calls to the SAT solver [4]. For a more extreme example, consider problem 25 from SYJ212, where *intuit* made 11,214 calls to the SAT solver and found a counter model with 1,955 worlds, whereas *intuitR* only made 45 calls and generated a counter model with only 4 worlds [4].

5 Conclusion

Intuitionistic Logic is a system of logic where all valid proofs are constructive. *intuitR* is a theorem prover that takes a propositional formula as its input and determines whether said formula is IPL-valid or not. It does so by first converting the formula into a special form, called an r-sequent, whose IPL-validity can be determined by continuously applying the two logical rules cpl_0 and cpl_1. *intuitR* does this in a double looping structure, where every iteration of the outer loop adds a new clause to the r-sequent by checking cpl_0 and applying cpl_1. The inner loop on the other hand attempts to find a useful clause to add while simultaneously constructing a counter model to the r-sequent in the process. *intuitR* was ran on a set of 1200 benchmark problems and its performance was compared to three other IPL provers. It was found to be able to solve 8 more problems than the next best solver and to have comparable speed to the fastest solver for most of the problem sets. There were four problem sets where *intuitR* was significantly faster than the other solvers and three where it was significantly slower.

While IPL might not necessarily have the most applications, the idea of utilizing a SAT solver to help in determining the validity of a formula in a non-classical logic system can be expanded to other logic systems. For example, it could be expanded beyond just IPL to include quantifiers for first-order intuitionistic logic. Claessen and Rosén describe how this might be done, as well as some potential problems with doing so in [2]. In [4], Fiorentini describes how a "reset" loop similar to the one *intuitR* contains can be expanded to other logics, giving Gödel-Dummett logic [8] as an example.

Acknowledgments

I would like to thank my advisor Professor Nic McPhee for his valuable guidance and insight throughout the process of writing this paper. I would also like to thank Professor Elena Machkasova and Dr. Stephen Adams for reading and providing feedback on previous drafts of this paper.

References

[1] Michael Barr. 1974. Toposes without points. *Journal of Pure and Applied Algebra* 5, 3 (1974), 265–280. https://doi.org/10.1016/0022-4049(74)90037-1

[2] Koen Claessen and Dan Rosén. 2015. SAT Modulo Intuitionistic Implications. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 622–637. https://doi.org/10.1007/978-3-662-48899-7_43

[3] Niklas Eén and Niklas Sörensson. 2004. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, Enrico Giunchiglia and Armando Tacchella (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 502–518. https://doi.org/10.1007/978-3-540-24605-3_37

[4] Camillo Fiorentini. 2021. Efficient SAT-based Proof Search in Intuitionistic Propositional Logic. In *Automated Deduction – CADE 28*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 217–233. https://doi.org/10.1007/978-3-030-79876-5_13

[5] Frank Van Harmelen, Vladimir Lifschitz, Bruce Porter, Carla P Gomes, Henry Krautz, Ashish Sabharwal, and Bart Selman. 2010. *Satisfiability Solvers.* Elsevier, 89–134.

[6] Samuel Mimram, Eric Goubault, Emmanuel Haucourt, Samuel Mimram, and Martin Raussen. 2021. *Program = proof.*

[7] Joan Moschovakis. 2021. Intuitionistic Logic. In *The Stanford Encyclopedia of Philosophy* (Fall 2021 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/entries/logic-intuitionistic/

[8] Jan von Plato. 2003. Skolem's Discovery of Gödel-Dummett Logic. *Studia Logica: An International Journal for Symbolic Logic* 73, 1 (2003), 153–157. http://www.jstor.org/stable/20016490