Prove-It: A Proof Assistant for Organizing and Verifying General Mathematical Knowledge

Wayne M. Witzel

Center for Computing Research, Quantum Computer Science
Sandia National Laboratories

Albuquerque, NM

wwitzel@sandia.gov

Warren D. Craft

Department of Computer Science,
The University of New Mexico

Albuquerque, NM

wdcraft@unm.edu

Robert D. Carr

Department of Computer Science,
The University of New Mexico

Albuquerque, NM

bobcarr@unm.edu

Joaquín E. Madrid Larrañaga

Center for Computing Research, Quantum Computer Science
Sandia National Laboratories

Albuquerque, NM

jmadri@sandia.gov

ABSTRACT

We introduce **Prove-It**, a Python-based general-purpose interactive theorem-proving assistant designed with the goal of making formal theorem proving as easy and natural as informal theorem proving (with moderate training). **Prove-It** uses a highly-flexible Jupyter notebook-based user interface that documents interactions and proof steps using LaTeX. We review Prove-It's highly expressive representation of expressions, judgments, theorems, and proofs; demonstrate the system by constructing a traditional proof-by-contradiction that $\sqrt{2} \notin \mathbb{Q}$; and discuss how the system avoids inconsistencies such as Russell's and Curry's paradoxes. Extensive documentation is provided in the appendices about core elements of the system. Current development and future work includes promising applications to quantum circuit manipulation and quantum algorithm verification.

KEYWORDS

automated theorem proving, proof assistant, python, Jupyter Notebook, quantum algorithm, quantum circuit

1 INTRODUCTION & MOTIVATION

Prove-It is an open-source Python library that has been designed as a general-purpose theorem prover to support structured proof [1] development to build formal proofs in a manner that mimics informal proof development. A primary target application for the Prove-It tool is the verification and analysis of quantum circuits and software, though it is designed as a general-purpose theorem prover. Quantum algorithm and circuit verification is a new but active area of research [2–8] as small quantum computing machines have become publicly available [9, 10]. The **Prove-It** approach is unique among the various other theorem provers [11] in its versatility, with a design intended to extend to any form of logical reasoning in any subject.

Flexibility has been a key design principle guiding the development of **Prove-It**, making it particularly well-suited to handle the various abstract concepts of quantum information theory. A prover for this application must not only support basic logical reasoning about propositional connectives and quantifiers over classical data values, including numbers and sets, but also express sophisticated concepts used in quantum computing, particularly linear algebra (matrices, tensor products, and unitary transformations) and probability theory (expectation values), as well as quantification over such constructs.

Another consideration is that the theorem prover is anticipated to be used by physicists and application experts in quantum computation who may not have training in formal methods. One of the main concerns in design decisions has thus been ease of proof construction (with sensible and predictable automation) and use of notation commonly used by physicists and mathematicians interested in scientific applications.

A guiding principle in the development of Prove-It has been to support the formalization of informal proofs (e.g.,

sketched in text books and journals) as a nearly direct translation. Ideally, with enough development in a particular subject area, it should be possible to develop formal proofs with no more difficulty than expressing an informal proof at a high level of abstraction. Most existing proof checkers [11, 12] are not likely to succeed at such proof attempts, either they do not support a sufficient level of sophisticated reasoning or gaps among various proof steps are too nontrivial to be automated. It is hoped that a user can formalize an informal proof at the same level of detail and fill the gaps in reasoning among various steps using additional separate but small proofs (which may be re-used for other purposes, and therefore the breadth of capabilities of Prove-It will grow over time). As an exercise, as well as a demonstration of a proof of this concept, a formal proof of the accuracy and bound of distribution of a quantum phase estimation algorithm was developed on a previous incarnation of Prove-It, mimicking the informal proof given in [13, pp 221–225]. This effort, while somewhat *ad hoc* and brute-force, involved 29 small proofs totalling 1750 steps by using over 275 assumptions, and was found to be extremely rewarding since it found 3 bugs in the informal proof in the book as well as managed to improve one of the bounds of the informal proof [14].

In the early development of **Prove-It**, we derived inspiration from the **The QED Manifesto** [15] and **The QED Manifesto Revisited** [12]. The former articulated the benefits to society that would derive from compiling a formalized computer database of all mathematical knowledge (the "QED system"). The latter explains why no such system exists to any great deal of satisfaction. It cites the lack of investment into a concerted effort to bring this about as well as the fact that (existing) formal systems fail to resemble standard mathematics (and instead look like computer code). We have made a concerted effort to address the latter issue in **Prove-It** by employing flexible LaTeX formatting of our formal expressions; while the LaTeX formatting merely serves as a superficial dressing on top of the actual formal representations, real mathematical notation presents a very intuitive interface that adds a great deal of value to the system. A suggested solution to the former issue is to create a 'Wikipedia for formalized mathematics' [12]. It is our hope that our open-source software will transition into a phase of rapid expansion, in proofs and proof strategies, through contributions from an increasing number of researchers that will mutually benefit by strengthening the system. This is an ambitious goal, but even if **Prove-It** does not succeed at becoming the "QED system," it would be worthwhile to inspire the system that does.

This paper is organized as follows. In §2, we have an important discussion regarding "domains of discourse." A distinguishing feature of the core logic of **Prove-It** is that the "domain of discourse" is never presumed or enforced by a type system; instead, domain restrictions are explicitly expressed in the *expressions* that form the *judgments* that form the *proofs*. §3 explores the "core derivation system" of **Prove-It** which refers to the information needed to understand and independently verify generated proofs and is intended to be fairly small and stable. §4 explores the theory package system, **Prove-It**'s ever-expanding library of knowledge. Layers of encapsulation, abstraction, and automation are constructed in this interdependent theory system; *theorems* may be proven from *axioms*, *conjectures* may be posited and used to construct partial proofs of other conjectures, and methods may be written to instantiate axioms/theorems/conjectures in a convenient manner. §5 demonstrates a proof that the square root of 2 is not rational, intended as an example that a formal proof construction can have a close resemblance to an informal proof. In §6, we address some questions about consistency and paradoxes. We conclude in §7. Appendices A-D provide a thorough guide to the core derivation system. Appendix E shows some of our basic theory packages and their axioms.

2 AN IMPORTANT NOTE REGARDING DOMAINS OF DISCOURSE

In deductive systems, it is typical to either presume a domain of discourse or use a type system which determines the type of any expression (term). In propositional logic [16], the domain of discourse is presumed to be the set of Boolean values (true or false which we denote as $\top$ and $\bot$ respectively). That is, every variable may be interpreted as being either true or false, and propositional formulas have a certain structure (well-formedness) to ensure they may only be true or false. In Zermelo–Fraenkel set theory[17], the domain of discourse is presumed to be sets. That is, every variable may be interpreted as being a set which is identified by its membership properties (*i.e.*, what elements does the set contain?). In a typed system, if x, y, and z are integers, $x \cdot y - z$ can be determined to be an integer. In contrast, **Prove-It** never presumes a domain of discourse and does not use a type system. Instead, domain restrictions in **Prove-It** must be expressed as explicit assumptions or conditions, or must be derived from explicit assumptions or conditions in order to be utilized.

To be more precise and concrete, let us first introduce *judgments* as the basic building blocks of any proof in our system. Borrowing from the language of natural deduction

proof calculi [18–21], a *judgment* is an object of knowledge, presented in the form

$$\{A_1, \ldots, A_n\} \vdash B \tag{1}$$

for some natural number n. The meaning of such a judgment is that B is true if A_1 through A_n are each true. "$\vdash$" is known as the turnstile symbol and represents logical entailment or deducibility. Essentially, if this judgment is true, it means that B follows logically if we assume that A_1 through A_n are each true. We call A_1 through A_n the assumptions of this judgment. We use curly braces to signify that the order and redundancy of the assumptions is unimportant (essentially acting as an enumerated, unordered set which is typically denoted with curly braces). When there are no assumptions (*i.e.*, $n = 0$), we simply write this with nothing on the left side of the turnstile. For example,

$$\vdash B \tag{2}$$

means that B can be derived to be true without any assumptions.

In the next section, we will discuss *expressions* as the building blocks of judgments and we will describe how a proof is constructed using the judgment building blocks. In this section, we focus on the meaning of the judgments themselves at a meta-logical level, in relation to domains of discourse in particular. In **Prove-It**, the assumptions (A_i) and consequent (B) do not need to be Boolean-valued. For the sake of the judgment, we only care about whether or not the parts of the judgment are true. We regard every expression, even if it contains gibberish, to be either equal to true or not equal to true.[1] The translation of $\{A_1, A_2\} \vdash B$ is quite literally, "we can derive B to be equal to true if we assume that A_1 and A_2 are both equal to true." This says nothing about A_1, A_2, and B having to be genuine propositions (unlike natural deduction systems, for example, in which you must first prove the parts of the judgment to be well-formed propositions before you can prove them to be true). If an assumption is used that is not a genuine proposition, this will simply be a matter of "garbage-in, garbage-out." What we care about is truth, and anything that is or can be legitimately true will necessarily be a perfectly legitimate proposition. Ultimately, this is only a matter of language, but our unique approach frees us from needing a type system and is always explicit about the domains of variables (via assumptions, conditions, or derivations).

Let us present some examples. Consider the "law of excluded middle." No problem. We simply need to be explicit about variables being in the Boolean domain (which we denote with $\mathbb{B}$).[2]

$$\{A \in \mathbb{B}\} \vdash A \vee \neg A \tag{3}$$

and

$$\{A \in \mathbb{B}, B \in \mathbb{B}\} \vdash (A \wedge B) \vee \neg(A \wedge B) \tag{4}$$

can each be proven, but the explicit assumptions are important. We can also prove harmless things about gibberish. For example,

$$\{x = (5 \vee \top), x + 10\} \vdash (5 \vee \top) + 10. \tag{5}$$

In this example, we are mixing types (5 and 10 are numbers and $\top$ is a Boolean) in a way that is meaningless to mathematicians and we are concluding something that is clearly not a proposition. It does not matter for our purpose of proving judgments. If we assume that the assumptions on the left are true statements, we can derive the consequent on the right. Garbage in, garbage out. If you find such garbage to be offensive, it would be easy, in principle, to filter it out. By allowing "garbage," however, we avoid the need of a type system that can feel clunky to those who are not experts in formal methods.

Our lack of implicit Boolean type restrictions extends to implications and conditions of quantifiers. In **Prove-It**, the implication $A \Rightarrow B$ means "B equals true if A equals true."[3] Therefore,

$$\vdash 3 \Rightarrow 3, \tag{6}$$

is a perfectly valid judgment that simply means 3 is equal to true if we assume it to be so. We could also prove $\vdash 3 \Rightarrow 5$ if we included axioms to prove that Booleans are distinct from numbers.[4] Furthermore, $\forall_{x \mid Q(x)} P(x)$ means "$P(x)$ is true for any x for which $Q(x)$ is true." Therefore,

$$\vdash \forall_{A \mid \neg A} \neg(A \wedge \top) \tag{7}$$

is valid even without an explicit restriction on the domain of A. Our quantifier has a condition $\neg A$ from which $A \in \mathbb{B}$ can be derived according to the current axioms of our system [Appendix E]. Specifically, we can only prove that $\neg A$ is true if A is false. It does not matter that $\neg 5$ is gibberish, only that $\neg 5 \neq \top$. Since "false" is the only value of A for which $\neg A$ is true, and since $\neg(A \wedge \top)$ is true if $A = \bot$, this is a provable and valid judgment by our definitions.

[1] Note, not being equal to true is not the same as being false. For example, the number 1 is not equal to true, but is not false either. Also, it may not always be possible to know whether or not any given expression is equal to true; that is, **Prove-It** is not a "complete" logic system and there is no intention to make it complete.

[2] See Table 2 for a list of logic and set theory symbols.

[3] $A \Rightarrow B$ has a similar meaning to the judgment $\{A\} \vdash B$. $\{A\} \vdash B$ may be derived from $\vdash A \Rightarrow B$ and vice-versa. However, $\{A\} \vdash B$ means that B is deducible when assuming A which depends upon the completeness of the theory while $A \Rightarrow B$ has a truth value that depends only upon truth values of A and B. Furthermore, we prove judgments, not implications (directly). An implication may be used within a judgment and may be nested within another implication, but a judgment may not be nested within another judgment.

[4] We are currently agnostic about whether or not Booleans and numbers are disjoint sets. We embrace agnosticism regarding things that are irrelevant.

3 BASIC LOGIC: THE CORE DERIVATION SYSTEM

Prove-It has a core system that is relatively small and intended to be fairly stable, and a theory package system that is an ever-expanding library of knowledge. The "core derivation system" refers to the information needed to understand and independently verify the proofs generated by **Prove-It**. The details of this system are provided in Appendices A, B, C, and D. In this section, we present the important, main concepts and provide useful examples.

3.1 Expressions and Proofs as DAGS

Prove-It uses objects, using Python's object-oriented language features, to represent *judgments* as they were described in §2. A judgment object is composed of *expression* objects to represent each assumption on the left of the turnstile and the one consequent expression on the right of the turnstile. Each judgment also has an associated proof object for deriving the judgment from ***axioms***, proven ***theorems***, and/or unproven ***conjectures*** that will be discussed later.

For example, in **Prove-It**'s `absolute_value` context, one can derive the following judgment:

$$\{x \in R^{\geq 0}, y \in R^{\geq 0}\} \vdash |x \cdot y| = (x \cdot y). \tag{8}$$

That is: assuming that x and y are non-negative real numbers, then the absolute value of their product is simply the product itself (without the absolute value).[5] This judgment is composed of three separate expressions: $x \in R^{\geq 0}$, $y \in R^{\geq 0}$, and $|x \cdot y| = (x \cdot y)$.

3.1.1 An Expression is a DAG. Mathematical expressions such as $|x \cdot y| = (x \cdot y)$, $x + y$, or $\forall_x P(x)$ are the basic building blocks of proofs and judgments. Each expression in **Prove-It** is represented internally by a directed acyclic graph (DAG) of expression and sub-expression objects.

Consider, for example, the mathematical expression $|x \cdot y|$ that appeared as part of judgment (8) above. Figure 1 shows how the input code and LaTeX-formatted output expression would look like in one of **Prove-It**'s interactive Python-based Jupyter notebooks.

In the Jupyter notebook, the user can click on the resulting LaTeX-formatted output expression $|x \cdot y|$ to inspect the DAG construction (in tabular form) representing the expression, as shown in Figure 2(a). That tabular summary corresponds to the DAG diagrammed in Figure 2(b), with the numerical notations next to each node indicating the corresponding rows of the **Prove-It** table.

```
1 Abs(Mult(x,y))
```

$|x \cdot y|$

Figure 1: Input code and LaTeX-formatted output in one of Prove-It's interactive Python-based Jupyter notebooks for the expression $|x \cdot y|$.

The DAG for the somewhat more complex expression $|a + b + 2 + 3| \in \mathbf{R}^{\geq 0}$ is shown in Figure (3) in both tabular and graphical forms. Where appropriate, the DAG edges have been labeled to describe the way in which one node contributes to the formation of a node just above. For example, the Literal + in node 7 serves as the operator on the four operands shown in node 8 to produce the summation operation shown in node 6. Those "operator" and "operand" notations correspond to the same notations appearing in the table version of the DAG (see row 6 of Figure 3(a)).

The "core type" columns of the Figure 2(a) and Figure 3(a) list one of nine primitive expression types for each node of the expression DAGs: **Variable**, **Literal**, **ExprTuple**, **Operation**, **Conditional**, **Lambda**, **NamedExprs**, **ExprRange**, and **IndexedVar**. These are described in Appendix A. Each of the nine primitive expression types has its own rules in the derivation system for building proofs and is therefore part of the "core derivation system" needed to independently verify the proofs generated by **Prove-It**. New expression classes may be derived from these nine (*e.g.*, `Abs` and `Mult` in Figure 1 are derived from **Operation** and provided with their own convenient methods and formatting options [see Appendix B]), but the derivation rules [described in Appendices C and D] are entirely dictated by their "core type" identities).

These nine primitive expression types provide for substantial expressivity. For example,

$$\forall_{m \in \mathbb{N}^+} \left[\forall_{A_1,\ldots,A_m \in \mathbb{B}} \ ((A_1 \vee \ldots \vee A_m) \in \mathbb{B})\right] \tag{9}$$

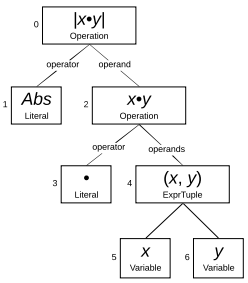
expresses that for any positive, whole number m, and any $A_1, \ldots, A_m$ that are each Boolean valued, their disjunction (logical or) is a Boolean value. Figure 4 shows the expression DAG for the sub-expression $\forall_{A_1,\ldots,A_m \in \mathbb{B}} \ ((A_1 \vee \ldots \vee A_m) \in \mathbb{B})$. Eight of the nine primitive expression types are used in this expression (all but **NamedExpr**). m is a **Variable**, **ExprRange**s and **IndexedVar**s are used to express $A_1, \ldots, A_m$ and $A_1 \vee \ldots \vee A_m$. The conditional quantifier

$$\forall_{A_1,\ldots,A_m \in \mathbb{B}} \ \ldots$$

[5] In the Jupyter Notebook-based user interface, and the **Prove-It** website [22], a user can then simply click on the turnstile $\vdash$ symbol to obtain a detailed listing of the steps in the proof of the judgment.

	core type	sub-expressions	expression
0	Operation	operator: 1 operand: 2	$\lvert x \cdot y \rvert$
1	Literal		Abs
2	Operation	operator: 3 operands: 4	$x \cdot y$
3	Literal		$\cdot$
4	ExprTuple	5, 6	(x, y)
5	Variable		x
6	Variable		y

(a)

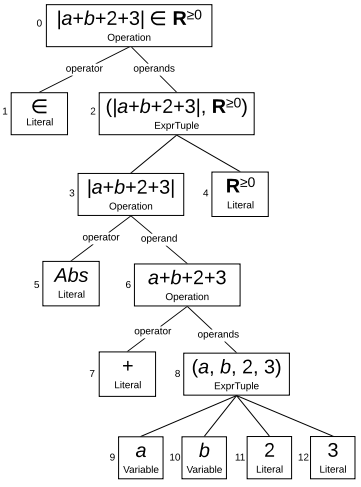


(b)

Figure 2: (*a*) **Prove-It**'s table representation of the directed acyclic graph (DAG) for the expression $|x\,y|$. (*b*) The corresponding graphical depiction of the DAG represented in part (*a*). The numerical notation to the left of each boxed node indicates the corresponding line in the expression table.

	core type	sub-expressions	expression
0	Operation	operator: 1 operands: 2	$\lvert a+b+2+3 \rvert \in \mathbb{R}^{\geq 0}$
1	Literal		$\in$
2	ExprTuple	3, 4	$\left(\lvert a+b+2+3 \rvert, \mathbb{R}^{\geq 0}\right)$
3	Operation	operator: 5 operand: 6	$\lvert a+b+2+3 \rvert$
4	Literal		$\mathbb{R}^{\geq 0}$
5	Literal		Abs
6	Operation	operator: 7 operands: 8	$a+b+2+3$
7	Literal		$+$
8	ExprTuple	9, 10, 11, 12	$(a, b, 2, 3)$
9	Variable		a
10	Variable		b
11	Literal		2
12	Literal		3

(a)



(b)

Figure 3: (*a*) **Prove-It**'s table representation of the directed acyclic graph (DAG) for the expression $|a+b+2+3| \in \mathbf{R}^{\geq 0}$. (*b*) A graphical depiction of the DAG shown in part (*a*). The numerical notation to the left of each boxed node indicates the corresponding line in the expression table.

is internally represented as an **Operation** with $\forall$ as a **Literal** operator and

$$(A_1, \ldots, A_m) \mapsto \ldots \text{ if } (A_1 \in \mathbb{B}) \wedge \ldots \wedge (A_m \in \mathbb{B})$$

as a **Lambda** operand. Within this **Lambda** operand is a **Conditional** (explicitly denoted with "if"). $(A_1, \ldots, A_m)$ is an **ExprTuple**. $\mathbb{B}$ is a **Literal**. And so on. While many of these primitive types are obscured by the flexible formatting, the structure that is revealed by the expression DAG with respect to primitive types is key to fully understanding **Prove-It** derivations.

Despite the apparent complexity of the expression, it is worth noting that the interactive user *construction* of such an expression is quite straightforward, with the actual **Prove-It** notebook code looking like this:

```
Forall(m,
       Forall(A_1_to_m,
              inBool(Or(A_1_to_m)),
              domain=Booleans),
       domain=NaturalsPos)
```

and utilizing nested `Forall` objects and an imported predefined expression `A_1_to_m` representing the $A_1, \ldots, A_m$.

	core type	sub-expressions	expression
0	Operation	operator: 1 operand: 2	$\forall_{A_1,\ldots,A_m \in \mathbb{B}}\ ((A_1 \vee \ldots \vee A_m) \in \mathbb{B})$
1	Literal		$\forall$
2	Lambda	parameters: 12 body: 3	$(A_1, \ldots, A_m) \mapsto \{(A_1 \vee \ldots \vee A_m) \in \mathbb{B} \text{ if } (A_1 \in \mathbb{B}) \wedge \ldots \wedge (A_m \in \mathbb{B})\,.$
3	Conditional	value: 4 condition: 5	$\{(A_1 \vee \ldots \vee A_m) \in \mathbb{B} \text{ if } (A_1 \in \mathbb{B}) \wedge \ldots \wedge (A_m \in \mathbb{B})\,.$
4	Operation	operator: 19 operands: 6	$(A_1 \vee \ldots \vee A_m) \in \mathbb{B}$
5	Operation	operator: 7 operands: 8	$(A_1 \in \mathbb{B}) \wedge \ldots \wedge (A_m \in \mathbb{B})$
6	ExprTuple	9, 22	$(A_1 \vee \ldots \vee A_m, \mathbb{B})$
7	Literal		$\wedge$
8	ExprTuple	10	$(A_1 \in \mathbb{B}, \ldots, A_m \in \mathbb{B})$
9	Operation	operator: 11 operands: 12	$A_1 \vee \ldots \vee A_m$
10	ExprRange	lambda_map: 13 start_index: 17 end_index: 18	$(A_1 \in \mathbb{B}), \ldots, (A_m \in \mathbb{B})$
11	Literal		$\vee$
12	ExprTuple	14	$(A_1, \ldots, A_m)$
13	Lambda	parameter: 24 body: 15	$\_a \mapsto (A_{\_a} \in \mathbb{B})$
14	ExprRange	lambda_map: 16 start_index: 17 end_index: 18	$A_1, \ldots, A_m$
15	Operation	operator: 19 operands: 20	$A_{\_a} \in \mathbb{B}$
16	Lambda	parameter: 24 body: 21	$\_a \mapsto A_{\_a}$
17	Literal		1
18	Variable		m
19	Literal		$\in$
20	ExprTuple	21, 22	$(A_{\_a}, \mathbb{B})$
21	IndexedVar	variable: 23 index: 24	$A_{\_a}$
22	Literal		$\mathbb{B}$
23	Variable		A
24	Variable		$\_a$

Figure 4: Prove-It's table representation of the directed acyclic graph (DAG) for the expression $\forall_{A_1,\ldots,A_m \in \mathbb{B}}\ ((A_1 \vee \ldots \vee A_m) \in \mathbb{B})$. The expression makes use of eight of **Prove-It**'s nine possible primitive expression types.

3.1.2 A Proof is a DAG. Like an expression, a ***proof*** or ***proof object*** in **Prove-It** is represented internally as a directed acyclic graph (DAG). For example, as part of Prove-It's `number/exponential` theory package, we have the following exponentiation theorem:

$$\vdash \forall_{a,x,y \in \mathbb{C}\,|\,x=y}\,(x^a = y^a)\,, \tag{10}$$

the proof of which appears in Figure 5 in both the table format produced in **Prove-It** and in a graphical representation of the underlying DAG.

Each node of a proof object DAG is a judgment. The subtree rooted at a node represents the proof of the judgment at that node. The leaf nodes are populated with assumptions, axioms, theorems, and conjectures.

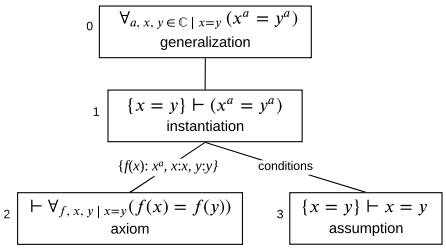
Proof DAGs are derivations that are presented in a reverse order from a usual derivation. This reversal reflects the tree-like structure of a proof where the root is the judgment being proven. It also offers a useful hierarchical view of a proof where you can easily focus on the important parts necessary to reach a conclusion, only digging into the details as needed. [6]

At the root node (labeled `0`) in Figure 5(*b*), for example, we have the exponentiation theorem being proven, along with the "step type" of "generalization" indicating how node

[6] The turnstile hyperlinks in the Jupyter Notebook-based user interface and the **Prove-It** website [22] are particular convenient for exploring proofs with this hierarchical view.

	step type	requirements	statement
0	generalizaton	1	$\vdash \forall_{a,x,y\in\mathbb{C}\ \mid\ x=y}\ (x^a = y^a)$
1	instantiation	2, 3	$\{x = y\} \vdash x^a = y^a$
	$f(x) : x^a,\ x : x,\ y : y$		
2	axiom		$\vdash \forall_{f,x,y\ \mid\ x=y}\ (f(x) = f(y))$
			proveit.logic.equality.substitution
3	assumption		$\{x = y\} \vdash x = y$

(a)



(b)

Figure 5: (*a*) **Prove-It**'s table representation of the directed acyclic graph (DAG) for the proof of the `exp_eq` theorem $\forall_{a,x,y\in\mathbb{C}\,\mid\,x=y}\ (x^a = y^a)$ (which appears in the `proveit.number.exponentiation` package). (*b*) A graphical depiction of the proof DAG represented in part (*a*). The number to the left of each boxed node indicates the corresponding line in the **Prove-It** proof table.

`0` is derived from nodes further down the tree. Contributing to the proof, we see at node 2 the general `substitution` axiom, and at node 3 the assumption that $x = y$. The edges are labeled to describe the way in which one node contributes to the derivation of the node above. For example, node 1 is derived by instantiating the substitution axiom at node 2 using the *instantiation (or replacement) map* $\{f(x) : x^a, x : x, y : y\}$, and using the assumption at node 3 to satisfies the $x = y$ condition required for this instantiation.

Similar to the "core type" identifications of an expression DAG, each node of the proof DAG is identified by a "step type." There are six possibilities corresponding to six inference rules described in Appendix C: **proof by assumption**, **axiom/theorem/conjecture invocation**, **modus ponens**, **deduction**, **instantiation**, and **generalization**. Among these, **instantiation**, which is used to instantiate universally quantified variables, is the most versatile and intricate, requiring additional expression-dependent reduction rules described in Appendix D. Any derivation step represented by a node in a proof DAG can be verified using an understanding of these inference and reduction rules as well as a knowledge of the expression DAG structures of the judgements of the node and its direct children.

The 'A' in DAG is very important. A valid proof must not contain circular logic in which a judgment (apart from leaf nodes) is proven using itself directly or indirectly. To make it easy to verify the acyclic nature of a proof DAG, requirements for a particular step will always appear after that step in the tree (*i.e.*, at a node further down or higher-numbered in the DAG). If multiple steps in the proof DAG have the same requirement, that requirement will appear after the last step that requires it. [7]

Theorem proofs will be discussed in the next section. For a theorem proof, we must not only ensure that the proof DAG is acyclic, we also need to ensure that there is no circularity in the dependencies among the theorems.

4 THE THEORY PACKAGE SYSTEM

The theory package system (TPS) of **Prove-It** is an ever-expanding library of knowledge, built upon a relatively small, stable foundational core. The TPS is a hierarchical collection of theory packages, each consisting of a collection of axioms, theorems/conjectures, and related proofs, organized around some common theme, and the packages already cover an array of topics (not entirely independent of one another). . Appendix E lists some of the fundamental theory packages of the current system and their axioms. For example, one very broad theory package in **Prove-It** is the `logic` package, inside of which appears the `sets` package, which itself contains a growing number of sub-packages devoted to topics such as `cardinality`, `membership`, `enumeration`, *etc.* The current theory package hierarchy is illustrated in Figure 6 (and can be accessed and interactively explored in depth at the **Prove-It** website [22]).

[7] For convenience in exploring proofs and sub-proofs (e.g., using turnstile hyperlinks), a proxy for all of the root node requirements will appear immediately following the root node, but some of these may simply reference duplicate judgments that are also required by steps further down (that is, instead of a regular "inference rule" step type, it will simply be marked by "reference" and have a single requirement for the later step with the same judgment). This makes it easy to view, together, all of the information that pertains to the root node derivation step while also enforcing the rule that requirements of a step always appear as a later step.

- proveit.logic
  - proveit.logic.booleans
    - proveit.logic.booleans.implication
    - proveit.logic.booleans.negation
    - proveit.logic.booleans.conjunction
    - proveit.logic.booleans.disjunction
    - proveit.logic.booleans.quantification
      - proveit.logic.booleans.quantification.universality
      - proveit.logic.booleans.quantification.existence
  - proveit.logic.equality
  - proveit.logic.sets
    - proveit.logic.sets.membership
    - proveit.logic.sets.enumeration
    - proveit.logic.sets.inclusion
    - proveit.logic.sets.unification
    - proveit.logic.sets.intersection
    - proveit.logic.sets.subtraction
    - proveit.logic.sets.comprehension
    - proveit.logic.sets.disjointness
    - proveit.logic.sets.cardinality
- proveit.numbers
  - proveit.numbers.sets
    - proveit.numbers.sets.naturals
    - proveit.numbers.sets.integers
    - proveit.numbers.sets.rationals
    - proveit.numbers.sets.real_numbers
    - proveit.numbers.sets.complex_numbers
  - proveit.numbers.numerals
    - proveit.numbers.numerals.binaries
    - proveit.numbers.numerals.decimals
    - proveit.numbers.numerals.hexidecimals
  - proveit.numbers.addition
    - proveit.numbers.addition.subtraction
  - proveit.numbers.negation
  - proveit.numbers.ordering
  - proveit.numbers.multiplication
  - proveit.numbers.division
  - proveit.numbers.modularity
  - proveit.numbers.exponentiation
  - proveit.numbers.summation
  - proveit.numbers.product
  - proveit.numbers.differentiation
  - proveit.numbers.integration

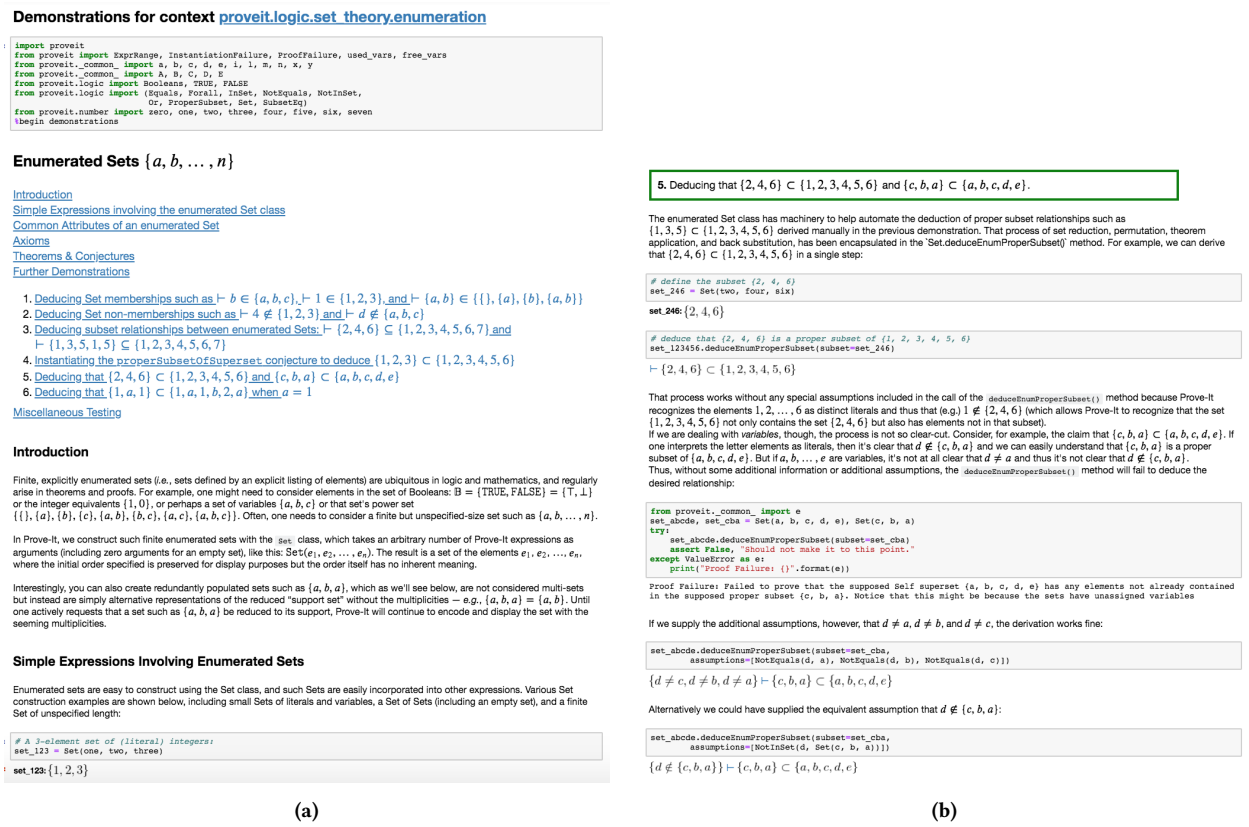
Figure 6: A snapshot of the current **Prove-It** theory package hierarchy, which can be accessed and interactively explored at www.pyproveit.org.

For convenience, each theory package also provides a collection of pre-defined common expressions and a "demonstrations" page to to document the features of the package. For example, the common expressions notebook for the `logic.booleans.quantification` theory package is shown in Figure 7, and the demonstrations page from the `logic.sets.enumeration` theory package is shown in Figure 8. The common expressions, axioms, theorems, and demonstrations pages are all Jupyter notebooks that display judgments/expressions/proofs using LaTeX-formatting. Each one is also available as an html web page on the **Prove-It** website [22]. Additionally, a theory package has python scripts which define expression classes and useful code for invoking theorems and performing various degrees of proof automation.

Figure 7: A screenshot of the "common expressions" notebook in **Prove-It**'s `logic.booleans.quantification` theory package, and an example of the many "common expressions" notebooks available throughout the **Prove-It** theory package system. Such notebooks provide useful predefined expressions that can be easily imported into proof notebooks, saving time and effort.

4.1 Axioms

An ***axiom*** in **Prove-It**, consistent with the commonly accepted meaning of the term, consists of a judgment of the form $\vdash C$, with no assumptions and no free variables, and which, by its definition and role, has no associated proof. As is common in mathematics and logic generally, axioms in **Prove-It** often provide the defining properties of various expression types, and can easily be "posited" by the **Prove-It** user through an easy, interactive construction process (either by adding axioms to an already-existing theory package or by creating an entirely new package within the **Prove-It** TPS).



(a) **(b)**

Figure 8: Screenshots of portions of the demonstrations notebook found in the `logic.sets.enumeration` theory package, showing (*a*) the table of contents and introductory remarks, and (*b*) one of the several demonstrations included in the notebook, in this case illustrating how one can define some enumerated sets and prove that one set is a proper subset of another under appropriate assumptions.

For example, the following transitivity, reflexivity, and symmetry axioms appear in the `logic/equality` context:

$$\vdash \forall_{x,y,z|x=y,y=z}[x = z], \tag{11}$$

$$\vdash \forall_{x}[x = x], \tag{12}$$

$$\vdash \forall_{x,y}[(y = x) = (x = y)], \tag{13}$$

and in the `logic/boolean/implication` context we have the axiomatic definition of logical equivalence:

$$\vdash \forall_{A,B}[(A \Leftrightarrow B) = ((A \Rightarrow B) \wedge (B \Rightarrow A))]. \tag{14}$$

As another example, a screenshot of the entire axiom notebook for the `logic.booleans.negation` package is shown in Figure 9.

Axioms will often appear as leaves in proof DAGs and can be explicitly imported into other package notebooks to support user explorations and interactive theorem-proving.

4.2 Theorems, Conjectures, and Dependencies

Theorems in **Prove-It** are similar in many ways to axioms. They also are judgments with no assumptions and no free variables, and they are grouped into theory packages. Unlike axioms, however, theorems need proofs. A theorem without a proof in the system is marked as a ***conjecture***. It may still be used in the proof of other theorems, but those theorems will also be marked as conjectures (in **Prove-It** theorem notebooks, such a conjecture is marked as a "conjecture with conjecture-based proof"). A conjecture converts to a theorem when it has a proof in the **Prove-It** system and the proof depends only upon axioms and other fully proven theorems (in a non-circular manner, of course). This easy availability of conjectures provides valuable flexibility, allowing top-down proof exploration and development. Users can generate interesting proofs based upon "conjectures" whose proofs can wait. Often, these are not conjectures in the true sense; rather, they are well-established facts that derive, in principle, from

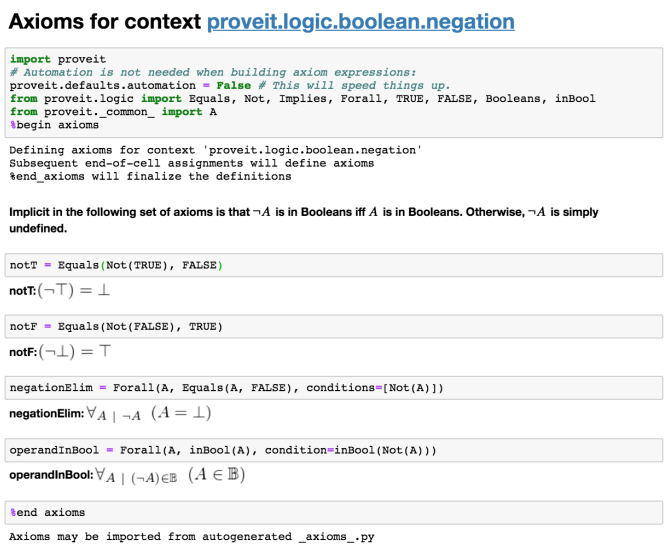


Figure 9: A screenshot of the axioms notebook in the `logic.booleans.negation` theory package. Axiom notebooks appear throughout the **Prove-It** theory package hierarchy at each package level, typically providing axiomatic definitions on which the package is based. See Appendix E for an extensive listing of package axioms.

proper axioms, but simply have not yet been proven in the **Prove-It** system. We use this feature a lot. It takes a substantial amount of work to build up a breadth of knowledge from only the most fundamental facts (axioms) and it can often be more valuable to prioritize theorem-proving of sophisticated theorems from which there is more insight to be gained. Of course, this feature could also be used for "authentic" conjectures (e.g., you could prove an algorithm complexity theorem based upon the conjecture that P $\neq$ NP).

Each theorem has an associated "proof" notebook (a Jupyter notebook in the interactive system or a web page in the **Prove-It** website [22]). The proof of the theorem may or may not be populated. When the proof is completed, the last cell will be a "%qed" command with the proof DAG as the output. An example of such a proof notebook for $\sqrt{2} \notin \mathbb{Q}$ is shown in §5. It may, however, depend upon other theorems/conjectures.[8]

While theorems/conjectures may be imported and instantiated directly in the construction of a proof (for a different theorem or a stand-alone proof), the best practice is to write methods in the Python scripts of the theory package that will instantiate these theorems indirectly. That way, the details of the theorem instantiation (the name of the theorem and choosing the expressions for instantiating each variable) are encapsulated in the package. Furthermore, **Prove-It** has useful automation features in which new facts are derived from other facts as they are created (e.g., when $S \vdash x = y$ is proven, $S \vdash y = x$ is automatically derived as a side-effect) or are automatically concluded as needed (e.g., given $S \vdash A$ and $T \vdash B$, $S \cup T \vdash A \wedge B$ can be concluded automatically). Ideally, constructing the proof then becomes, after some experience and gaining familiarity with the theory packages involved, as natural as writing an informal proof by hand but with certainty of correctness and less tedious writing. Our example in §5 has many instances of theorems being instantiated indirectly, either through full automation (see the cells that simply call the `prove` method) or convenient methods that perform directed transformations.

4.2.1 Theorem dependencies form a DAG. Just as there must not be any circular dependency within a particular proof object, there must not be any circular dependency among the various theorem proofs. That is, a theorem may have a proof that depends upon other theorems whose proofs depend upon other theorems, and so forth. Individual proofs may also depend upon axioms and unproven conjectures. These dependencies form a graph that must be acyclic to avoid circular logic at the theorem level. The leaf nodes are axioms or conjectures. With a particular theorem at the root, when there are only axioms as leaf nodes and no conjectures, that theorem is said to be "fully proven" and is no longer marked as conjecture.

Because theorems may be proven in any order and may be invoked indirectly via automation, **Prove-It** requires a mechanism to constrain which theorems are "usable" in any given theorem proof. For example, say that it is intended for theorem B to depend upon theorem A but neither have been proven. However, automation has already been developed for automatically instantiating theorem B as needed. Furthermore, assume that this automation would allow theorem A to be trivially but erroneously proven as a consequence of theorem B. That would create a problem because if we let theorem A be proven using theorem B as a dependency, then theorem B would not be allowed to use theorem A as intended (since the dependencies must be acyclic). Our solution is to explicitly indicate the theorems and packages that may be *presumed* within each theorem proof. In cell 2 of our $\sqrt{2} \notin \mathbb{Q}$ demonstration of §5, shows an example of "presuming" theory packages.[9]

[8] It is worth noting that **Prove-It** makes no distinction between a theorem and a lemma. What a user might consider a lemma, **Prove-It** considers just another theorem with no special status, although the Jupyter notebook user-interface will allow the user to include contextual comments and explanations if desired.

[9] We intend to make our "presuming" machinery more convenient for users/developers, so these details are subject to change.

For a conjecture with a proof, it is useful to know what dependent conjectures remain to be proven for it to gain the "fully proven" status. For any theorem/conjecture with a proof, it is useful to know what axioms are required (as leaves of the dependency DAG). After all, it is easy to add axioms, so it is a good idea to make sure that the axioms used in a proof of interest are trusted.[10] Going the other direction, it can also be useful or interesting to know all of the theorems that depend upon an axiom or theorem. All of this information is presented on "dependency" notebooks (a Jupyter notebook in the interactive system or a web page in the **Prove-It** website [22]). There are hyperlinks to `<dependencies>` on each "proof" notebook as well as the "expression information" notebooks that may be reached by clicking on any axiom / theorem / conjecture expression. Stripped down contents of the "dependency" notebook for the $\sqrt{2} \notin \mathbb{Q}$ example is shown in §5.1 (it excludes the names of the axioms / conjectures that appear in the actual "dependency" notebook and is displayed in a more compact form).

5 EXAMPLE PROBLEM: $\sqrt{2} \notin \mathbb{Q}$

In this section we present a "theorem proof" notebook for $\vdash \sqrt{2} \notin \mathbb{Q}$. This is a nice example that is not too trivial but not terribly complicated. Furthermore, we can compare our demonstration with a convenient compilation of this same demonstration implemented in 17 other theorem provers [11]. Two additional proofs stand out among the others as easily comprehensible by anybody who is sufficiently well-versed in mathematics: a geometrical proof and an informal proof. However, these are not implemented in a formal language that can be verified through automation. In the formal proof demonstrations, the strategies employed to generate the proof are not obvious without being an expert in that particular system.

Our proof notebook is shown in the following pages. As with all Jupyter notebooks, there are input cells (labeled "`In[#]`") and corresponding output cells (labeled "`Out[#]`"). The inputs show our commands for constructing the proof and the outputs show the progress along the way (usually a proven judgment whose proof is accessible via the hyperlinked $\vdash$ symbol). Additionally, there are "markdown" cells that describe the intentions as we work through the proof. The final cell issues a `%qed` command followed by the output of the generated proof DAG. Our proof has generated, largely through computer automation, a proof DAG with 143 steps (nodes), but we only show a few of these in this paper for brevity. Some of these steps invoke theorems/conjectures. Currently, this proof still has a "conjecture" status since we have not proven all of the dependent conjectures. The list of dependent conjectures/axioms (the leaf nodes of the dependency DAG) is shown in §5.1.

We hope that our proof stands out among the $\sqrt{2} \notin \mathbb{Q}$ proofs of other formal systems [11] as being nearly as straightforward and readable as the informal proof. The output judgments are easily readable by anybody familiar with the mathematical notation. The proof DAG is similarly readable with some basic understanding of our inference rules [Appendix C]. Also, it is not hard to deduce the purpose and effect of the input commands, given the "markdown" comments and the output. Therefore, one does not need to be a **Prove-It** expert in order to understand a proof construction and proof DAG in a **Prove-It** notebook. Constructing a proof does require a certain amount of expertise and practice, but it certainly helps to have comprehensible demonstrations as examples; many others can be found on the **Prove-It** website [22]. Automation features and convenient methods for applying other theorems/conjectures of the system are also tremendously valuable. This example demonstrates a number of these features/methods. Our ultimate goal is to make it as easy (or easier) to construct a formal proof as it is to construct an informal but rigorous one. With sufficient theory package development, the complexity of the mathematics should not be a barrier to this goal.

[10] To really be certain about a proof, one should examine the expression DAG structure of each axiom as well as the proven theorem.

Proof of proveit.number.exponentiation.sqrt2_is_not_rational theorem

In [1]:
```
import proveit
from proveit import defaults, Variable
from proveit._common_ import p, a_star, b_star
from proveit.logic import InSet, Equals
from proveit.number import (zero, two, NaturalsPos, Rationals, RationalsPos,
                            Greater, Mult, frac, Divides, GCD, Exp, sqrt)
```

In [2]:
```
%proving sqrt2_is_not_rational presuming [proveit.logic, proveit.number]
```

```
Beginning proof of sqrt2_is_not_rational
Recorded 'presuming' information
Presuming theorems in proveit.logic, proveit.number (except any that presume
this theorem).
Presuming previous theorems (applied transitively).
```

Out[2]: **sqrt2_is_not_rational:** $\sqrt{2} \notin \mathbb{Q}$

(see dependencies)

Setting up a contradiction proof by first assuming $\sqrt{2} \in \mathbb{Q}$

In [3]:
```
sqrt_2_in_rationals = InSet(sqrt(two), Rationals)
```

Out[3]: **sqrt_2_in_rationals:** $\sqrt{2} \in \mathbb{Q}$

In [4]:
```
defaults.assumptions = [sqrt_2_in_rationals]
```

Out[4]: **defaults.assumptions:** $\left(\sqrt{2} \in \mathbb{Q}\right)$

If $\sqrt{2}$ is in $\mathbb{Q}$, then it must be in $\mathbb{Q}^+$

In [5]:
```
from proveit.number import RealsPos
InSet(sqrt(two), RealsPos).prove()
```

Out[5]: $\vdash \sqrt{2} \in \mathbb{R}^+$

In [6]:
```
Greater(sqrt(two), zero).prove()
```

Out[6]: $\vdash \sqrt{2} > 0$

In [7]:
```
sqrt_2_in_rationals_pos = InSet(sqrt(two), RationalsPos).prove()
```

Out[7]: **sqrt_2_in_rationals_pos:** $\left\{\sqrt{2} \in \mathbb{Q}\right\} \vdash \sqrt{2} \in \mathbb{Q}^+$

Choose relatively prime a^* and b^* such $\sqrt{2} = a^*/b^*$ assuming $\sqrt{2} \in \mathbb{Q}$

(Relatively prime means there are no nontrivial common divisors; that is, $gcd(a^*, b^*) = 1$ where gcd is the "greatest common divisor")

In [8]:
```
sqrt_2_in_rationals_pos.choose_reduced_rational_fraction(a_star, b_star)
```

```
Creating Skolem 'constant(s)': (a_star, b_star).
Call the KnownTruth.eliminate(a_star, b_star) to complete the Skolemization
(when the 'constant(s)' are no longer needed).
Adding to defaults.assumptions:
```

Out[8]: $\left(a^* \in \mathbb{N}^+, b^* \in \mathbb{N}^+, \left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*, b^*) = 1)\right)$

From $\sqrt{2} = a^*/b^*$, derive $2 \cdot (b^*)^2 = (a^*)^2$

In [9]:
```
sqrt_2_equation = Equals(sqrt(two), frac(a_star, b_star)).prove()
```

Out[9]: **sqrt_2_equation:** $\left\{\left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*, b^*) = 1)\right\} \vdash \sqrt{2} = \frac{a^*}{b^*}$

In [10]:
```
sqrt_2_equation = sqrt_2_equation.right_mult_both_sides(b_star, simplify=True)
```

Out[10]: **sqrt_2_equation:**

$$\left\{b^* \in \mathbb{N}^+, \sqrt{2} \in \mathbb{Q}, \left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*, b^*) = 1), a^* \in \mathbb{N}^+\right\} \vdash \left(\sqrt{2} \cdot b^*\right) = a^*$$

In [11]:
```
sqrt_2_equation = sqrt_2_equation.square_both_sides()
```

Out[11]: **sqrt_2_equation:**

$$\left\{a^* \in \mathbb{N}^+, b^* \in \mathbb{N}^+, \sqrt{2} \in \mathbb{Q}, \left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*, b^*) = 1)\right\} \vdash \left(\sqrt{2} \cdot b^*\right)^2 = (a^*)^2$$

In [12]:
```
sqrt_2_equation = sqrt_2_equation.innerExpr().lhs.distribute()
```

Out[12]: **sqrt_2_equation:**

$$\left\{a^* \in \mathbb{N}^+, b^* \in \mathbb{N}^+, \sqrt{2} \in \mathbb{Q}, \left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*, b^*) = 1)\right\} \vdash \left((\sqrt{2})^2 \cdot (b^*)^2\right) = (a^*)^2$$

In [13]:
```
two_bStar_sqrd__eq__aStar_sqrd = sqrt_2_equation.innerExpr().lhs.simplify()
```

Out[13]: **two_bStar_sqrd__eq__aStar_sqrd:**

$$\left\{a^* \in \mathbb{N}^+, b^* \in \mathbb{N}^+, \sqrt{2} \in \mathbb{Q}, \left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*, b^*) = 1)\right\} \vdash \left(2 \cdot (b^*)^2\right) = (a^*)^2$$

Now prove that $2 \mid a^*$ via $2 \cdot (b^*)^2 = (a^*)^2$

In [14]:
```
InSet(Exp(b_star, two), NaturalsPos).prove()
```

Out[14]: $\left\{b^* \in \mathbb{N}^+\right\} \vdash (b^*)^2 \in \mathbb{N}^+$

Proving $2 \mid (2 \cdot (b^*)^2)$ is trivial knowing $(b^*)^2 \in \mathbb{N}^+$

In [15]:
```
two_divides_two_bStar_sqrd = Divides(two, Mult(two, Exp(b_star, two))).prove()
```

Out[15]: **two_divides_two_bStar_sqrd:** $\left\{b^* \in \mathbb{N}^+\right\} \vdash 2 \mid \left(2 \cdot (b^*)^2\right)$

In [16]:
```
two_bStar_sqrd__eq__aStar_sqrd.subRightSideInto(two_divides_two_bStar_sqrd)
```

Out[16]:

$$\left\{b^* \in \mathbb{N}^+, a^* \in \mathbb{N}^+, \sqrt{2} \in \mathbb{Q}, \left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*, b^*) = 1)\right\} \vdash 2 \mid (a^*)^2$$

$2 \mid (a^*)$ was proven as a side-effect of $2 \mid (a^*)^2$ since any integer is even if its square is even

In [17]:
```
two_divides_aStar = Divides(two, a_star).prove()
```

Out[17]: **two_divides_aStar:**

$$\left\{a^* \in \mathbb{N}^+, b^* \in \mathbb{N}^+, \sqrt{2} \in \mathbb{Q}, \left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*, b^*) = 1)\right\} \vdash 2 \mid a^*$$

Next derive $2 \mid b^*$ from $2 \cdot (b^*)^2 = (a^*)^2$ and $2 \mid (a^*)$ in a few steps

Derive $2^2 \mid (a^*)^2$ from $2 \mid (a^*)$

In [18]:
```
two_sqrd__divides__aStar_sqrd = \
    two_divides_aStar.introduce_common_exponent(two)
```

Out[18]: **two_sqrd__divides__aStar_sqrd:**

$$\left\{a^* \in \mathbb{N}^+, b^* \in \mathbb{N}^+, \sqrt{2} \in \mathbb{Q}, \left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*, b^*) = 1)\right\} \vdash 2^2 \mid (a^*)^2$$

Derive $2^2 \mid 2 \cdot (b^*)^2$ from $2^2 \mid (a^*)^2$

In [19]:
```
two_sqrd__divides__two_bStar_sqrd = \
    two_bStar_sqrd__eq__aStar_sqrd.subLeftSideInto(
        two_sqrd__divides__aStar_sqrd)
```

Out[19]: **two_sqrd__divides__two_bStar_sqrd:**

$$\left\{a^* \in \mathbb{N}^+, b^* \in \mathbb{N}^+, \sqrt{2} \in \mathbb{Q}, \left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*, b^*) = 1)\right\} \vdash 2^2 \mid \left(2 \cdot (b^*)^2\right)$$

Derive $2 \mid (b^*)^2$ from $2^2 \mid 2 \cdot (b^*)^2$

In [20]:
```
two_sqrd__eq__two_times_two = Exp(two, two).expansion()
```

Out[20]: **two_sqrd__eq__two_times_two:** $\vdash 2^2 = (2 \cdot 2)$

In [21]:
```
two_sqrd__eq__two_times_two.subRightSideInto(
    two_sqrd__divides__two_bStar_sqrd)
```

Out[21]:

$$\left\{a^* \in \mathbb{N}^+, b^* \in \mathbb{N}^+, \sqrt{2} \in \mathbb{Q}, \left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*, b^*) = 1)\right\} \vdash (2 \cdot 2) \mid \left(2 \cdot (b^*)^2\right)$$

In [22]:
```
Divides(two, Exp(b_star, two)).prove()
```

Out[22]:

$$\left\{b^* \in \mathbb{N}^+, a^* \in \mathbb{N}^+, \sqrt{2} \in \mathbb{Q}, \left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*, b^*) = 1)\right\} \vdash 2 \mid (b^*)^2$$

Finally, derive $2 \mid (b^*)$ from $2 \mid (b^*)^2$

In [23]:
```
two_divides_bStar = Divides(two, b_star).prove()
```

Out[23]: **two_divides_bStar:**

$$\left\{b^* \in \mathbb{N}^+, a^* \in \mathbb{N}^+, \sqrt{2} \in \mathbb{Q}, \left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*, b^*) = 1)\right\} \vdash 2|b^*$$

Prove a contradiction given that $2 \mid (a^*)$, $2 \mid (b^*)$, and a^* and b^* were chosen to be relatively prime

In [24]:
```
aStar_bStar_relatively_prime = GCD(a_star, b_star).deduce_relatively_prime()
```

Out[24]: **aStar_bStar_relatively_prime:**

$$\left\{a^* \in \mathbb{N}^+, b^* \in \mathbb{N}^+, \left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*, b^*) = 1)\right\} \vdash \forall_{p \in \mathbb{N}^+ \mid p>1} \left(\neg\left((p|a^*) \wedge (p|b^*)\right)\right)$$

In [25]:
```
two_does_not_divide_both = aStar_bStar_relatively_prime.specialize({p:two})
```

Out[25]: **two_does_not_divide_both:**

$$\left\{a^* \in \mathbb{N}^+, b^* \in \mathbb{N}^+, \left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*, b^*) = 1)\right\} \vdash \neg\left((2|a^*) \wedge (2|b^*)\right)$$

In [26]:
```
contradiction_with_choices = two_does_not_divide_both.deriveContradiction()
```

Out[26]: **contradiction_with_choices:**

$$\left\{a^* \in \mathbb{N}^+, b^* \in \mathbb{N}^+, \sqrt{2} \in \mathbb{Q}, \left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*, b^*) = 1)\right\} \vdash \bot$$

Use the existence of a^* and b^*, assuming $\sqrt{2} \in \mathbb{Q}$, to eliminate extra assumptions (effective Skolemization)

In [27]:
```
contradiction = contradiction_with_choices.eliminate(a_star, b_star)
```

Out[27]: **contradiction:** $\left\{\sqrt{2} \in \mathbb{Q}\right\} \vdash \bot$

Prove $\sqrt{2} \notin \mathbb{Q}$ via contradiction

In [28]:
```
sqrt2_impl_F = contradiction.asImplication(hypothesis=sqrt_2_in_rationals)
```

Out[28]: **sqrt2_impl_F:** $\vdash \left(\sqrt{2} \in \mathbb{Q}\right) \Rightarrow \bot$

In [29]:
```
sqrt2_impl_F.denyAntecedent(assumptions=[])
```

Out[29]: $\vdash \neg\left(\sqrt{2} \in \mathbb{Q}\right)$

In [30]:
```
%qed
```

Out[30]:

	step type	requirements	statement
0	instantiation	1, 2	$\vdash \sqrt{2} \notin \mathbb{Q}$
	$x : \sqrt{2}, S : \mathbb{Q}$		
1	theorem		$\vdash \forall_{x,S \mid \neg(x \in S)}\ (x \notin S)$
			proveit.logic.set_theory.membership.foldNotInSet
2	instantiation	3, 4, 5, 6	$\vdash \neg\left(\sqrt{2} \in \mathbb{Q}\right)$
	$A : \sqrt{2} \in \mathbb{Q}, B : \bot$		
3	theorem		$\vdash \forall_{A \in \mathbb{B}}\ \left[\forall_{B \mid A \Rightarrow B, \neg B}\ (\neg A)\right]$
			proveit.logic.boolean.implication.modusTollensDenial
4	instantiation	7	$\vdash \left(\sqrt{2} \in \mathbb{Q}\right) \in \mathbb{B}$
	$x : \sqrt{2}$		
5	deduction	8	$\vdash \left(\sqrt{2} \in \mathbb{Q}\right) \Rightarrow \bot$
6	theorem		$\vdash \neg\bot$
			proveit.logic.boolean.negation.notFalse
7	conjecture		$\vdash \forall_{x}\ ((x \in \mathbb{Q}) \in \mathbb{B})$
			proveit.number.sets.rational.xInRationalsInBool
8	modus ponens	9, 10	$\left\{\sqrt{2} \in \mathbb{Q}\right\} \vdash \bot$
9	instantiation	11, 12, 13, 75	$\vdash \left(\left[\exists_{a,b \in \mathbb{N}^+}\ \left(\left(\sqrt{2} = \frac{a}{b}\right) \wedge (gcd(a,b) = 1)\right)\right] \wedge \left[\forall_{a^*,b^* \in \mathbb{N}^+}\ \left(\left(\left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*,b^*) = 1)\right) \Rightarrow \bot\right)\right]\right) \Rightarrow \bot$

$n : 2, P(a,b) : \left(\sqrt{2} = \frac{a}{b}\right) \wedge (gcd(a,b) = 1), Q(a,b) : \left(a \in \mathbb{N}^+\right) \wedge \left(b \in \mathbb{N}^+\right), \alpha : \bot, (y_1, \ldots, y_2) : (a,b), (x_1, \ldots, x_2) : (a^*, b^*)$

10 instantiation 29, 14, 15 $\left\{\sqrt{2} \in \mathbb{Q}\right\} \vdash \left[\exists_{a,b \in \mathbb{N}^+} \left(\left(\sqrt{2} = \frac{a}{b}\right) \wedge (gcd(a,b) = 1)\right)\right] \wedge \left[\forall_{a^*,b^* \in \mathbb{N}^+} \left(\left(\left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*,b^*) = 1)\right) \Rightarrow \bot\right)\right]$

$A : \exists_{a,b \in \mathbb{N}^+} \left(\left(\sqrt{2} = \frac{a}{b}\right) \wedge (gcd(a,b) = 1)\right), B : \forall_{a^*,b^* \in \mathbb{N}^+} \left(\left(\left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*,b^*) = 1)\right) \Rightarrow \bot\right)$

11 conjecture $\vdash \forall_{n \in \mathbb{N}^+} \left[\forall_{P,Q,\alpha} \left(\left(\left[\exists_{y_1,\ldots,y_n \mid Q(y_1,\ldots,y_n)} P(y_1,\ldots,y_n)\right] \wedge \left[\forall_{x_1,\ldots,x_n \mid Q(x_1,\ldots,x_n)} (P(x_1,\ldots,x_n) \Rightarrow \alpha)\right]\right) \Rightarrow \alpha\right)\right]$

proveit.logic.boolean.quantification.existential.skolemElim

12 instantiation 16 $\vdash |(a^*, b^*)| = |(1, \ldots, 2)|$

$a : a^*, b : b^*$

13 instantiation 16 $\vdash |(a,b)| = |(1, \ldots, 2)|$

$a : a, b : b$

14 instantiation 17, 18 $\left\{\sqrt{2} \in \mathbb{Q}\right\} \vdash \exists_{a,b \in \mathbb{N}^+} \left(\left(\sqrt{2} = \frac{a}{b}\right) \wedge (gcd(a,b) = 1)\right)$

$q : \sqrt{2}$

15 generalizaton 19 $\left\{\sqrt{2} \in \mathbb{Q}\right\} \vdash \forall_{a^*,b^* \in \mathbb{N}^+} \left(\left(\left(\sqrt{2} = \frac{a^*}{b^*}\right) \wedge (gcd(a^*,b^*) = 1)\right) \Rightarrow \bot\right)$

16 conjecture $\vdash \forall_{a,b} \left(|(a,b)| = |(1, \ldots, 2)|\right)$

proveit.number.numeral.deci.tuple_len_2_typical_equiv

17 conjecture $\vdash \forall_{q \in \mathbb{Q}^+} \left[\exists_{a,b \in \mathbb{N}^+} \left(\left(q = \frac{a}{b}\right) \wedge (gcd(a,b) = 1)\right)\right]$

proveit.number.sets.rational.reducedNatsPosRatio

... **steps omitted** ...

143 assumption $\left\{b^* \in \mathbb{N}^+\right\} \vdash b^* \in \mathbb{N}^+$

5.1 Dependencies for our $\sqrt{2} \notin \mathbb{Q}$ proof

Unproven conjectures required (directly or indirectly) to prove `sqrt2_is_not_rational`

- $\forall_{i \in \mathbb{N}} \left[\forall_{f} \ (|\ (f(1), \ldots, f(i))\ | = |\ (1, \ldots, i)\ |)\right]$
- $\forall_{a,b \in \mathbb{N}^+ \mid gcd(a,b)=1} \left[\forall_{p \in \mathbb{N}^+ \mid p>1} \ (\neg\ ((p|a) \wedge (p|b)))\right]$
- $\forall_{k \in \mathbb{Z}, a \in \mathbb{Z}, n \in \mathbb{N}^+ \mid k|a} \ (k^n | a^n)$
- $\forall_{a,b,k \in \mathbb{C} \mid (k \cdot a)|(k \cdot b), k \neq 0} \ (a|b)$
- $\forall_{a,n \in \mathbb{Z} \mid 2|a^n} \ (2|a)$
- $\forall_{x \in \mathbb{C}, y \in \mathbb{Z} \mid x \neq 0} \ (x|\ (x \cdot y))$
- $\forall_{x \in \mathbb{C}} \ \left(\frac{x}{1} = x\right)$
- $\forall_{a,b,c,d,e \in \mathbb{C} \mid c \neq 0} \ \left(\left(\frac{a}{b \cdot c} \cdot \frac{c \cdot e}{d}\right) = \left(\frac{a}{b} \cdot \frac{d}{e}\right)\right)$
- $\forall_{a \in \mathbb{N}^+, b \in \mathbb{N}} \ \left(a^b \in \mathbb{N}^+\right)$
- $\forall_{n \in \mathbb{N}^+, x \in \mathbb{R}^+} \ \left((x^{\frac{1}{n}})^n = x\right)$
- $\forall_{a \in \mathbb{C}, b \in \mathbb{C}, n \in \mathbb{N}^+} \ ((a \cdot b)^n = (a^n \cdot b^n))$
- $\forall_{a \in \mathbb{R}^+} \ \left(\sqrt{a} \in \mathbb{R}^+\right)$
- $\forall_{x \in \mathbb{C}} \ \left(x^2 = (x \cdot x)\right)$
- $\forall_{x \in \mathbb{C}} \ ((1 \cdot x) = x)$
- $\forall_{x \in \mathbb{C}} \ ((x \cdot 1) = x)$
- $\forall_{a,x,y \in \mathbb{C} \mid x=y} \ ((x \cdot a) = (y \cdot a))$
- $1 < 2$
- $2 \in \mathbb{N}^+$
- $\forall_{a,b} \ (|\ (a, b)\ | = |\ (1, \ldots, 2)\ |)$
- $\forall_{a \in \mathbb{R}^+} \ (a > 0)$
- $\mathbb{R} \subset \mathbb{C}$
- $\mathbb{N} \subset \mathbb{Z}$
- $\mathbb{N}^+ \subset \mathbb{N}$
- $\forall_{n \in \mathbb{N}^+} \ (n \neq 0)$
- $\mathbb{Z} \subset \mathbb{Q}$
- $\forall_{q \in \mathbb{Q} \mid q>0} \ (q \in \mathbb{Q}^+)$
- $\forall_{q \in \mathbb{Q}^+} \ \left[\exists_{a,b \in \mathbb{N}^+} \ \left(\left(q = \frac{a}{b}\right) \wedge (gcd(a, b) = 1)\right)\right]$
- $\forall_{x} \ ((x \in \mathbb{Q}) \in \mathbb{B})$
- $\mathbb{N}^+ \subset \mathbb{R}^+$
- $\mathbb{Q} \subset \mathbb{R}$

Axioms required (directly or indirectly) to prove `sqrt2_is_not_rational`

- $(\top \wedge \top) = \top$
- $\forall_{A \mid A=\top} A$
- $\forall_{A \mid A} \ (A = \top)$
- $\forall_{A \in \mathbb{B} \mid (\neg A) \Rightarrow \bot} A$
- $\forall_{A \in \mathbb{B} \mid A \Rightarrow \bot} \ (\neg A)$
- $(\neg \bot) = \top$
- $(\neg \top) = \bot$
- $\forall_{n \in \mathbb{N}^+} \left[\forall_{P,Q} \left(\begin{array}{c} \left[\exists_{x_1,\ldots,x_n \mid Q(x_1,\ldots,x_n)} \ P(x_1, \ldots, x_n)\right] = \\ \neg \left[\begin{array}{c} \forall_{y_1,\ldots,y_n \mid Q(y_1,\ldots,y_n)} \\ (P(y_1, \ldots, y_n) \neq \top) \end{array} \right] \end{array} \right)\right]$
- $\forall_{x,y} \ ((x = y) \in \mathbb{B})$
- $\forall_{x,y} \ ((y = x) = (x = y))$
- $\forall_{x,y,z \mid x=y, y=z} \ (x = z)$
- $\forall_{x,y} \ ((x \neq y) = (\neg\ (x = y)))$
- $\forall_{f,x,y \mid x=y} \ (f(x) = f(y))$
- $\forall_{A,B} \ \left((A \subset B) = \left((A \subseteq B) \wedge \left(B \ncong A\right)\right)\right)$
- $\forall_{A,B} \ ((A \subseteq B) = [\forall_{x \in A} \ (x \in B)])$
- $\forall_{x,S} \ ((x \notin S) = (\neg\ (x \in S)))$
- $1 = (0 + 1)$
- $2 = (1 + 1)$
- $\forall_{x,y} \ ((y > x) = (x < y))$
- $\forall_{n \in \mathbb{N}} \ ((n + 1) \in \mathbb{N})$
- $0 \in \mathbb{N}$

6 AVOIDANCE OF KNOWN PARADOXES

At this time, we do not have a full consistency proof for the logic and basic axioms of **Prove-It**. Even before this is achieved, **Prove-It** can be a valuable tool. Its proofs are designed to be human-readable and well-structured, and therefore should be at least as acceptable as very rigorous proofs generated by hand that are the standard in much of the literature. That is, you can take its proofs at face value for experts to decide if the arguments, specific to the proof, are sound. Still, one would hope that it is not possible to derive

$$\vdash \bot$$

using standard axioms of **Prove-It**. A consistency proof would serve the purpose of arguing that this is not possible. While we are not prepared to claim that we can rigorously prove this to be the case just yet, we are prepared to discuss some specific well-known logical fallacies and how **Prove-It** does not fall into their traps.

6.1 Russell's paradox (what is a set?)

What is a set? What are sets allowed to contain? If the definition is too broad, one is vulnerable to Russell's paradox [23]. Consider, as Bertrand Russell did, defining the set of all sets that do not contain themselves. Call this set R. Now ask, is $R \in R$? If we assume $R \in R$, then we can prove that $R \notin R$ since R does not contain any set that contains itself. If we assume $R \notin R$, then we can prove that $R \in R$ since R contains all sets that do not contain themselves. Thus, we cannot define R to be a set (in which membership is clearly defined) without generating a contradiction.

There are multiple ways to define sets and avoid Russell's paradox. In choosing our axioms [Appendix E], we essentially follow the standard approach of ZFC set theory [24]. First, we only allow a restricted form of set comprehension,

known as specification or separation in set theory literature [17]. Unrestricted set comprehension allows one to define sets according to any criterion for membership (e.g., "Let S be the set of all x such that $Q(x)$ is true."). Russell's set is defined in this manner. Restricted set comprehension only allows us to define subsets of existing sets in this manner (e.g. "Let $S \subseteq T$ be the set of all $x \in T$ such that $Q(x)$ is true."). Second, all sets have a well-defined rank that is an ordinal number [25]. Loosely speaking, the rank of a set is the depth to which sets are nested; the empty set has rank zero, and a set containing a set containing the empty set has a minimum rank of 2. A set that contains itself cannot have a definite rank (the rank, r, would have to be larger than r which is not possible for an ordinal number). Furthermore, a set cannot contain a superset of itself, either directly or indirectly (as a member of a member, etc.) for the same reason. In other words, the universe of sets (which is *not*, itself, a set), is the von Neumann universe, also known as the cumulative hierarchy of sets [17].

Box 1:
Avoiding Russel's paradox with irreflexive set membership

Define Russell's set with restricted set comprehension in the following way: given any set T, we define $R_T = \{S \in T \mid S \notin S\}$. There are two conditions that any given S must satisfy in order to belong to Russell's set, R_T. First, S must be an element of T. Second, S must satisfy the condition that $S \notin S$.

Define a set T to be "easy" if:

(1) $\forall_{S \in T}\,[S \notin S]$.
(2) $\forall_{S \subseteq T}\,[S \notin S]$.
(3) $T \notin T$ (redundant from 2).

We can prove that, if T is easy, then $R_T = T$ (*i.e.*, all members of R_T are members of T and vice-versa).

For all $S \in T$, $S \in R_T$ iff $S \notin S$. R_T is a subset of T. Define the set $S' = R_T$ and we assume for now that S' is eligible to be in R_T. Then $(S' \in R_T)$ iff $(S' \notin S')$. Therefore, $(S' \in R_T)$ iff $(S' \notin R_T)$, and finally $(R_T \in R_T)$ iff $(R_T \notin R_T)$. This is a contradiction.

Assume that set membership is never reflexive (a set cannot contain itself). Then all sets are "easy" and we can show that S' is not eligible to be in R_T so we don't get a contradiction. Since T is easy, for any $S \in T$ we have the true statement $S \notin S$. Therefore $R_T = T$. But $T \notin T$ because T is easy. Hence $S' = R_T = T$ is not eligible to be in R_T so there is no Russell contradiction.

Prohibiting unrestricted set comprehension and ensuring that membership is irreflexive (that is, no set may be a member of itself) will prevent Russell's paradox. Using restricted set comprehension, one may only define Russell's sets as parameterized by a superset T: $R_T = \{x \in T \mid x \notin x\}$. In this notation, R_T is the subset of all elements of T which do not contain themselves. If sets are not allowed to contain themselves (*i.e.*, set membership is irreflexive), $R_T = T$.[11] This irreflexive property will be maintained if we assume that T, and all of its members, are themselves irreflexive with respect to membership. (See Box 1 for some more detail.)

Ensuring membership is irreflexive (along with restricting set comprehension) is sufficient but not necessary to avoid Russell's paradox (for example, it is possible to allow sets that only contain themselves, known as Quine atoms [26]). Ensuring that sets have a definitive, ordinal rank is sufficient but may not be necessary to ensure that membership is irreflexive. The rank property, however, is useful for making meta-logical arguments regarding the consistency of axioms. Using an inductive argument to prove that a given property is maintained by the axioms of a theory, the rank property is a more useful (stronger) induction hypothesis than the irreflexive membership property. In any case, ZFC is a standard foundation of mathematics, so it is wise to follow its conventions for defining sets. At a meta-logical level for determining and analyzing the axioms of **Prove-It**, we therefore choose our sets to be defined to have a definitive, ordinal rank.[12]

The main difference between our approach and ZFC set theory is that we never implicitly presume domains of discourse as we discussed in §2. ZFC set theory presumes a domain of discourse of sets. Every variable in ZFC represents a set. In contrast, we use conditions that directly or indirectly involve set membership/non-membership when we want to restrict a corresponding domain to that of sets. As long as our axioms only define membership/non-membership to be true for "valid" sets,[13] proper restrictions will be imposed with such conditions. Importantly, no axiom states that $(x \in S)$ is a Boolean in general for any S. It is only defined to be a Boolean if S is a "valid" set, or is equivalent to a "valid" set with respect to set equivalence.

As an example, consider the following judgment that can be derived from our restricted comprehension axiom (axiom 1 of `proveit.logic.sets.comprehension` in Appendix E):

$$\vdash \forall_{S,Q,x} \left(\begin{array}{c} \left(x \in \{y \mid Q(y)\}_{y \in S}\right) \\ = \left[\exists_{y \in S \mid Q(y)}\, x = y\right] \end{array} \right). \qquad (15)$$

[11] More precisely, without making implicit assumptions about equivalent sets being equal, $R_T \cong T$, meaning that $\forall_x (x \in R_T) = (x \in T)$.

[12] Our axioms may, however, be agnostic (incomplete) with respect to defining the rank of certain sets. Unlike ZFC, our sets may contain elements that are not known to be sets and therefore have an unknown rank. That is fine as long as the axioms are consistent with the existence of a well-defined rank for each set which ensures that sets do not directly or indirectly contain themselves or supersets of themselves.

[13] Here, "valid set" is a meta-logical concept meaning that the set has a definite, ordinal rank.

We may use this to define any set of the form $\{y \mid Q(y)\}_{y \in S}$ as the subset of S that satisfies the condition defined by Q. There is no explicit restriction on S; however, if this is instantiated with any S which is not equivalent to a "valid set," it will not be possible to prove whether or not $x \in S$ is true for any x. Therefore, you will not be able to prove that anything is or is not contained in your gibberish "set." If one attempted to create a Russell-type set, R_T, with a T that is not a proper set, one would *not* be able to prove that anything is contained in R_T. It would be consistent with assuming $T \cong R_T \cong \emptyset$. Therefore, no contradiction would be possible. Furthermore, while there is no explicit restriction on Q, its role in the condition of the existential quantifier ensures that we may interpret (at a meta-logical level) $Q(y)$ as $Q(y) = \top$. Thus, while there is no restriction to ensure that Q be instantiated with a function whose domain is S and codomain is $\mathbb{B}$, it is only relevant that it map elements of S to objects that are equal to true or not equal to true (which may include anything since we assume that every expression is either equal to true or not equal to true, though it is not always knowable which one it is).

We are not claiming to have a rigorous proof that we avoid Russell's paradox at this time. We believe that we do avoid it (and that **Prove-It**, with its basic axioms, is more broadly consistent), but there are tricky details that would be involved with such a proof. Not only do we impose set domain restrictions indirectly through membership definitions, we also do not have explicit constraints on functions (they need not even be proper "functions" in the set-theoretic sense of having a specific domain and co-domain). We will discuss this in the next section about Curry's paradox.

6.2 Kleene-Rosser-Curry's paradox (what is a function?)

What is a function? A definition that is too broad leaves one vulnerable to Curry's paradox.[14] The standard definition of a mathematical function (coming from ZFC set theory) is that it is a relation (ordered pair) between two sets (a domain and a codomain) in which there is a unique output (in the codomain) for every input (in the domain) [24]. This definition is considered to be "safe," but **Prove-It** does, in general, relax the requirement with respect to specifying a set-theoretic domain and codomain. Such restrictions can be made [*e.g.*, $f \in \text{Map}(\mathbb{A}, \mathbb{B})$ could be defined as $\forall_{x \in \mathbb{A}}\, f(x) \in \mathbb{B}$, effectively imposing $\mathbb{A}$ as the domain of f and $\mathbb{B}$ as its codomain]. However, we have chosen some of our axioms to allow nonrestrictive "functions." For example, the substitution axiom [see axioms of `proveit.logic.equality` in Appendix E],

$$\vdash \forall_{f,x,y \mid x=y}\ (f(x) = f(y)), \tag{16}$$

allows us to substitute (replace) any x for any y in *any* expression provided that $x = y$, regardless of whether or not f has a domain containing x and y.

Russell's paradox led to a contradiction by defining a set in a self-referential manner. Applying a function in a self-referential manner can also lead to a paradox, as we will see by using Curry's paradox. The paradox can be expressed in various logic systems [31] without proper restrictions in place. Most relevant for **Prove-It**'s purpose are the forms of the paradox in untyped lambda calculus and combinatory logic where it relates to the notion of a "function." But let us start with the informal version that is easiest to explain. Begin with a sentence of the form "If this sentence is true, then the moon is made of cheese" (analogous to examples in [31]). Using "reasonable" arguments, we can derive "the moon is made of cheese" even though this is untrue. Now use the law of deduction (if we can prove B assuming A, then $A \Rightarrow B$). In this case, we take "this sentence is true" as the assumption A and use it to prove "the moon is made of cheese" as the consequent B [since $A = (A \Rightarrow B)$ by definition]. In so doing, we have proven $A \Rightarrow B$ which is the original sentence. Since A represents our original sentence being true, we have thus proven A and can then derive B (for any B, true or not).

In order to prove a contradiction using Curry's paradox, the analogue to "this sentence is true" must be constructed. We adapt a form of Curry's paradox from [28] to illustrate how **Prove-It** (we believe) avoids this pitfall. Define g as

$$P \mapsto (P(P) \Rightarrow \bot) \tag{17}$$

using the notation of **Prove-It**.[15] Using beta reduction, it can be proved that

$$\vdash g(g) = (g(g) \Rightarrow \bot). \tag{18}$$

Note that this has the same form as $A = (A \Rightarrow B)$. With a few derivation steps, this will lead to a contradiction. It is easy to see that $g(g) = (g(g) \Rightarrow \bot)$ is an equation with no solution. $g(g)$ is either equal to true or not (as are all implications in **Prove-It**). If it is not equal to true, we have a contradiction because the right side will be true (in **Prove-It**'s logic, an

[14] Curry's paradox was discovered by Haskell Curry when he attempted to extend combinatory logic with logical connectives in order to include the extensionality axiom: two combinator expressions e_1 and e_2 are equivalent iff for every x, $e_1(x) = e_2(x)$. Kleene and Rosser [27] discovered a similar paradox in Church's extended system that also combined untyped lambda calculus with logical connectives using Richard's paradox (see an excellent article by Rosser [28] explaining the background of the paradox). Curry came up with a cleaner version using Russell's paradox [29, 30].

[15] In classical untyped lambda calculus, $g = (\lambda P.(P(P) \Rightarrow A)$ and can be constructed using the fixed point Y combinator since $g(g) = g$.

implication is always true when the antecedent is not true). But if it is equal to true, we can prove $\top \Rightarrow \bot$ and thus $\bot$ (a contradiction). This is similar to having an equation such as $x = x + 1$ in which there is no solution.

If we could create the expression of g applied to itself without introducing an axiom, we would be in trouble. That is, if we could express
$[P \mapsto (P(P) \Rightarrow A)](P \mapsto (P(P) \Rightarrow A))$ as an **Operation** expression where the *operator* and *operand* are the same **Lambda** expression, we'd be able to create an expression that is intrinsically paradoxical.[16] This is like Russell's paradox in which the expression $\{x \mid x \notin x\}$ is intrinsically paradoxical in naïve set theory. Fortunately, our axioms do not admit an actual definition for $\{x \mid x \notin x\}$ (as we saw in Sec. 6.1). Similarly, our expression syntax does not allow $[P \mapsto (P(P) \Rightarrow A)](P \mapsto (P(P) \Rightarrow A))$ to be constructed.[17]

It is possible, in **Prove-It**, to create $g(g)$ as an **Operation** expression with g as a **Variable**. We could then instantiate $g : P \mapsto (P(P) \Rightarrow \bot)$. However, **Prove-It**, according to the reduction rules described in Appendix D, will then greedily apply beta reduction until there is no more beta reduction to be performed. In this case, however, that process never terminates. So, for practical purposes, $g(g)$ with $g : P \mapsto (P(P) \Rightarrow \bot)$ cannot be expressed in **Prove-It** without including an assumption or adding an erroneous axiom that defines $g = [P \mapsto (P(P) \Rightarrow \bot)]$.

At a meta-logical level, we can regard **Prove-It** functions to have an implicit domain restriction that disallows any input that results in non-terminating beta reductions. A function should be defined to have a single, definite evaluation given any particular input. When beta reductions fail to terminate, it suggests that there is no single, definitive evaluation [e.g., $g(g) = (g(g) \Rightarrow \bot)$ has no definitive evaluation as we saw]. With this implicit domain restriction imposed on all functions, enforced by **Prove-It**'s core rules and consistent with our basic axioms, we believe that Curry's paradox is avoided. We admit that an implicit restriction of the domain of a function according to whether or not the reduction of the function application terminates is a nonstandard approach. But it does appear to be effective, at least in regards to avoiding Curry's paradox.

[16]The analogue to this (a lambda expression as an operator) in lambda calculus is lambda application with lambda abstraction as the first term. This is allowed in lambda calculus systems but not in **Prove-It**. Untyped lambda calculus, when extended with logical connectives, can be vulnerable to Curry's paradox for this reason.

[17]For this same reason, the fixed-point (Y) combinator of combinatory logic cannot be constructed either.

7 DISCUSSION & CONCLUSIONS

We introduce **Prove-It**, a Python-based general-purpose interactive theorem-proving assistant designed with the goal to make formal theorem proving as easy and natural as informal theorem proving. The Jupyter notebook-based interface and underlying Python code make the **Prove-It** system accessible to a wide audience, moderately easy to learn and use, and relatively easy to make contributions to its expandable knowledge base of axioms, conjectures, theorems, proofs, and proof tactics.

Prove-It expressions, proofs (that may depend upon theorems and conjectures), and proof dependencies are all represented internally by directed acyclic graphs (DAGs). Proof and proof dependency DAGs help organize a proof result into a human-readable presentation and ensure against circular logic. Expression DAGs make the connection between internal representations used by the core logical system and mathematical notation formatted elegantly in LaTeX. By supporting the use of conjectures, sophisticated and interesting theorems may be tentatively proven, dependant upon obvious or commonly accepted facts before they are fully proven in the system. Details of our system are provided in the appendices.

As an example, we show a proof of $\sqrt{2} \notin \mathbb{Q}$ which demonstrates many of **Prove-It**'s features. We show how, with the use of automation and methods (proof tactics) that encapsulate axiom/theorem/conjecture invocation, the construction of a formal proof in **Prove-It** does, in fact, resemble an informal proof. We demonstrate the use of conjectures that allow us to present such a demonstration before we have had the chance to prove some basic, simple facts that are needed for this proof.

We present arguments in support of the logical consistency of **Prove-It**, addressing two well known potential pitfalls in particular: Russell's paradox and Curry's paradox. We believe that we avoid these pitfalls but do acknowledge that we lack a rigorous consistency proof at this time. Our system uses nonstandard approaches in its avoidance of requiring type theory that merits further investigation by experts in mathematical foundations. However, because our proofs are presented in an intuitive, human-readable form, the value of the system is not substantially diminished by the lack of complete certainty regarding its consistency. Our system makes it relatively easy to generate formal, structured proofs, with no gaps in the chain of reasoning, that may be examined at face value by experts who can decide whether or not a given proof is convincing.

In addition to ongoing development and expansion of fundamental theory packages in logic, set theory, numerical sets, *etc*, current development and future work involves the application of **Prove-It** to issues of quantum computation and quantum circuit verification and design. As a quick preview of such work, Figure 10 shows an example quantum circuit (representing a quantum teleportation algorithm) generated in the **Prove-It** system, and Figure 11 shows a quantum-circuit-based expression of a general theorem in **Prove-It** for replacing an equivalent portion of a valid quantum circuit of arbitrary size. Future development may also include interfacing with a SAT or SMT solver and generally strengthening its automation capabilities.

8 ACKNOWLEDGMENTS

We gratefully acknowledge Deepak Kapur, Kenny Rudinger, Mohan Sarovar, Jon Aytac, Geoffrey Hulette, Denis Bueno, and Geoffrey Reedy for valuable discussions and contributions to the development of **Prove-It**.

This work was supported by the the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under the Quantum Computing Applications Team (QCAT) program, and the Laboratory Directed Research and Development program at Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

```
Circuit(ExprArray(ExprTuple(Input(ket_psi), IdentityOp(), IdentityOp(), MultiQubitGate(CONTROL, Set(one, two)),
                            Gate(H), MES, MultiQubitGate(CLASSICAL_CONTROL, Set(one, three)), SPACE),
                  ExprTuple(Input(ket0), IdentityOp(), MultiQubitGate(X, Set(two, three)).withStyles(representation='implicit'
                            MultiQubitGate(X, Set(one, two)).withStyles(representation='implicit'),
                            IdentityOp(), MultiQubitGate(Literal('MES'), Set(two, three)), SPACE, SPACE),
                  ExprTuple(Input(ket0), Gate(H), MultiQubitGate(CONTROL, Set(two, three)), IdentityOp(), IdentityOp(),
                            MultiQubitGate(X, Set(two, three)), MultiQubitGate(Z, Set(one, three)), Output(ket_psi)))).withSty
```

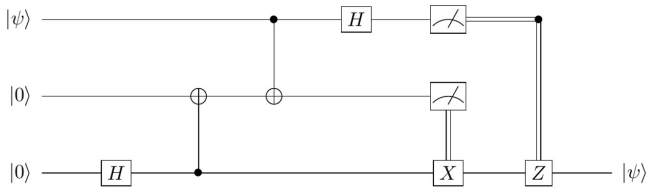


Figure 10: An example of a quantum circuit generated using Prove-It (in this case, representing a quantum teleportation algorithm).

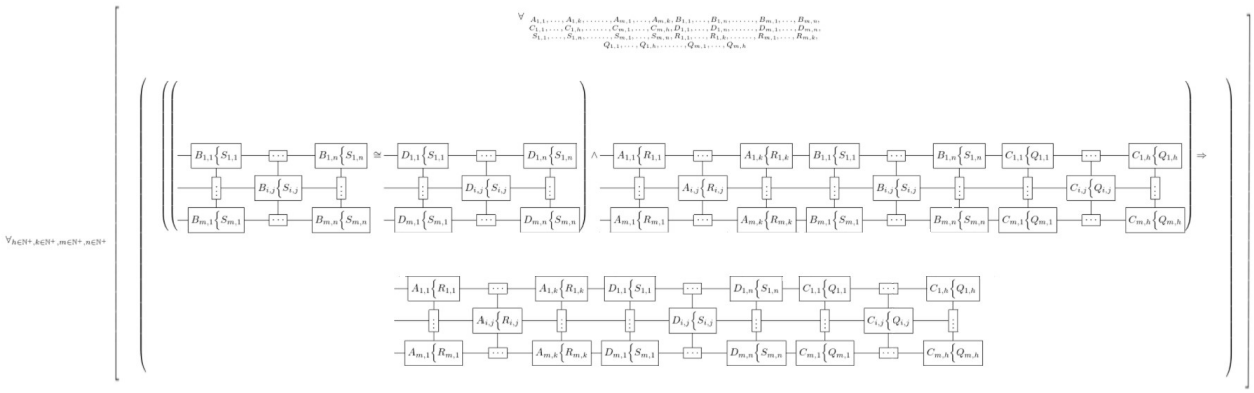


Figure 11: Our time_equiv_judgement theorem that allows us to replace an equivalent portion of a valid (true) circuit of arbitrary size. The theorem states that for any valid quantum circuit, labeled in a manner that expresses a true identity analogous to an equation, the replacement of a portion of this circuit with an equivalent circuit of the same size results in another valid circuit. This is done with the use of generic multi-qubit gates and Prove-It's ExprRanges.

Appendices

A CORE/PRIMITIVE EXPRESSION CLASSES

Prove-It uses an object-oriented framework for defining different types (classes) of expressions. Each expression class defines how the expression is to be formatted (into LaTeX or as a character string) and provides convenient methods for applying axioms or theorems/conjectures pertinent to the particular expression. Each expression class must derive from on of the following primitive expression types that are defined in the core of **Prove-It** (the `proveit._core_` package):

Variable A label that is interchangeable (as long as it is kept distinct from different labels) with no intrinsic meaning. It is often represented by a single letter (a, b, x, y, etc.) but can have any representation.

Literal A label that is not interchangeable and has an intrinsic meaning. Specific operators ($\neg$, $\wedge$, $+$, $\times$, etc.) and specific irreducible values ($\top$, $\bot$, 0, 5, etc.) are all **Literal**s. Furthermore, a problem-story "variable" in a particular theory package, representing some unknown but particular value, should also be a **Literal** (e.g., "Ann has a apples...").

ExprTuple Represents a finite, ordered list (sequence) of elements that may be used when there are multiple *operands* of an **Operation** (see next), or multiple *parameters* of a **Lambda** (see below). An **ExprTuple** contains zero or more *entries* as sub-expressions. An entry may represent a single element of the mathematical tuple being represented, or it may be an **ExprRange** (see below) that represents any number of mathematical elements. For example,
$(a, b, c_1, ...c_n)$
has three *entries* with the third entry being an **ExprRange** that represents n elements of the mathematical tuple.

Operation The application of an *operator* on *operand(s)*. For example, $0 + 5 + 8$, $A \vee B \vee C \vee D$, and $x < y$ are examples of *operation* expressions. The *operator* must be a **Literal**, **Variable**, or an **IndexedVar**. They may *not* be **Lambda** expressions. This is an important restriction that prevents Curry's paradox (discussed in Sec. 6.2). The **Prove-It** theory packages define many types derived from the **Operation** type (e.g., for each specific operation), but all operations have the same behavior with respect to reduction rules that will be discussed in Appendix D.

Conditional An expression that has one *condition* and one *value*. The **Conditional** axiomatically reduces (equates) to its *value* if and only if the *value* is true. For example $\{P(x) \text{ if } Q(x).$ reduces to $P(x)$ if $Q(x)$ is true. Otherwise, the **Conditional** is not defined (i.e., it is not reducible). Furthermore, we have the following axiom
$$\vdash \forall_{a,Q} \left(\begin{array}{c} \{a \text{ if } Q\ . \\ = \{a \text{ if } Q = \top\ . \end{array} \right)$$
which defines the **Conditional** only up the the truth of its *condition* with no regard, beyond this, to whether or not the *condition* has a Boolean value. This is consistent with the manner in which **Prove-It** judgments are treated without type restrictions.

Lambda A mapping of *parameter(s)* into a *body* that may involve any or all of the *parameters*. For example,
$(x, y, z) \mapsto \{x + y/z \text{ if } x, y, z \in \mathbb{R}, z \neq 0.$
is represented by a **Lambda** with a **Conditional** *body* that converts three real numbers x, y, z to $x + y/z$ as long as z is not zero. Each parameter represented by the *parameter(s)* must be a **Variable** or **IndexedVar** (discussed below). The **Lambda** introduces a new scope for the *parameters*. The *parameters* are said to be bound in this new scope; occurrences outside this scope are not deemed to be the same thing. **Lambda** expressions that are equivalent up to a relabeling of the parameters are deemed by **Prove-It** to be the *same* expression (or sub-expression); for example,
$(a, b, c) \mapsto \{a + b/c \text{ if } a, b, c \in \mathbb{R}, c \neq 0.$
is deemed to be the same expression as the one above, simply using a, b, c instead of x, y, z. This is known as alpha conversion in the lambda calculus terminology and is discussed in Appendix B.2 in more detail.

NamedExprs A mapping from keyword strings to expressions. This can be used to prevent ambiguity of an expression's internal representation when the order of a sequence of sub-expressions is not enough to specify the role of the sub-expression.

ExprRange Represents a range of expressions with a *parameter* going from a *start_index* to an *end_index* in successive unit increments (+1). It contains a **Lambda** sub-expression (*lambda_map*) to define each expression in the range as a function of the *parameter* value (the index). For example, $1/(x+i)+...+1/(x+j)$ is represented by an **Operation** with '+' as the *operator* and an **ExprRange** as the *operand*. The **ExprRange** sub-expression is $1/(x+i)$, ..., $1/(x+j)$. The sub-expressions of this exprtypeExprRange may be denoted as
lambda_map : $k \mapsto 1/(x + k)$
start_index : i
end_index : j

IndexedVar A special kind of **Operation** that indexes a **Variable** with one or more indices. It treats the variable being indexed as the *operator* and the *index* or *indices* as the *operand(s)*. For example: a_1, x_i, x_{i+1}, $x_{i,j}$, etc. An **ExprRange** of **IndexedVar**s acts as a collection of variables and may be used in an **ExprTuple** of **Lambda** *parameters*. For example: $(a_1, \ldots, a_n, b_{1,1}, \ldots, b_{1,k}, \ldots\ldots, b_{j,1}, \ldots, b_{j,k})$ forms a valid collection of **Lambda** *parameters* with the b variables being doubly indexed and contained in doubly-nested **ExprRange**s.

These primitive expression types are special for the following reason. In order to rigorously verify the correctness of a proof in **Prove-It**, one must know the DAG structure of the expressions involved with respect these primitive types. Additionally, $\forall$ (forall), $\Rightarrow$ (implies), $\wedge$ (conjunction), and $=$ (equals) **Literal**s play specific roles in the derivation and reduction rules. However, the derived expression classes are not important in this verification process (e.g., one need only know that $x+y$ is an **Operation** with $+$ as a **Literal** operator and x and y as **Variable** operands, not that $x + y$ is an `Add` type expression). This is important for the sake of having a relatively small and stable *core derivation system* for proof verification.

Classes are derived from these primitives for the purpose of defining specific ways of formatting expressions (as a string or as LaTeX) and to provide methods for convenience and automation for manipulating expressions of that specific type, but this is external to the *core derivation system* for proof verification. The most common primitive type used as a base class is the **Operation**; there are a wide variety of kinds of operations (logical connective operations such as $\wedge$, $\vee$, $\neg$, etc.; the $\forall$ and $\exists$ quantifiers, number operations such as $+$, $-$, etc.). Each one of these has its own methods that are convenient for applying axioms and theorems specific to the operation.

A few non-primitive expression classes are defined in the core (`proveit._core_`) for organizational purposes. These are:

OperationOverInstances An **Operation** whose *operand* is a **Lambda** expression. Often, the **Lambda** expression has a **Conditional** body. The idea is that it operates over the domain of instances of the **Lambda** parameter variables for which the *condition* of the **Conditional** is satisfied. Quantifiers are **OperationOverInstances**. $\forall_{x,y \mid Q(x,y)} P(x,y)$ is internally represented by **Prove-It** as an operation of the $\forall$ operator acting on $(x, y) \mapsto \{P(x,y) \text{ if } Q(x,y)$.

ConditionalSet An **Operation** that operates on any number of **Conditional** expressions. If one and only one of the *condition* expressions is satisfied (proven true), and the rest are proven to be untrue (though not necessarily false), it axiomatically reduces (equates) to the *value* associated with the satisfied *condition*. For example, consider this definition for the absolute value of an integer:
$$\begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}.$$
This would be represented by a **ConditionalSet** with two **Conditional** *operands*.

ExprArray An **ExprTuple** of **ExprTuple**s formatted as a 2-dimensional array. It may contain **ExprRange**s. For example,
$$\begin{array}{cccccc} A_i & \cdots & A_j & B_i & \cdots & B_j \\ C_i & \cdots & C_j & D_i & \cdots & D_j \end{array}$$
is an **ExprArray** that is a tuple of two tuples, each containing ranges of indexed variables, formatted in a "horizontal" layout style.

B EXPRESSION STYLE CHOICE

As far as the **Prove-It** software is concerned, the meaning of an expression is entirely dictated by the expression DAG with respect to the primitive base classes. The actual formatting of an expression (in LaTeX or as a text string) is a separate matter and there are no safeguards to ensure that the formatted expression reflects its meaning. Of course, in developing **Prove-It** we do our best to make sure that formatting routines will make the notation clear and unambiguous. But by not placing restrictions on the formatting aspect, we have a lot of freedom in rendering elegant mathematical representations in LaTeX, and can offer flexibility in the style choices of this formatting. To be certain about the genuine meaning of an axiom or a theorem, users are encouraged to view the full expression DAG. The elegant LaTeX representations, however, are extremely convenient to aid one's intuition about the mathematical constructs being utilized.

B.1 Style options

Each expression class has its own style options which can be modified to change the formatting of an expression without changing its meaning. For example,
a/b and $\frac{a}{b}$
are both manifestations of the same expression of the `Div` class defined in the `proveit.numbers.division` theory package. The first one has its `division` style set to `inline`; the second one has its `division` style set to `fraction`. A few other examples were already mentioned in Appendix A.

$\forall_{x \mid Q(x) \wedge R(x)} P(x)$	$\forall_{x \mid Q(x), R(x)} P(x)$
$\{\}$	$\emptyset$
$A \subseteq B$	$B \supseteq A$
$A \subset B$	$B \supset A$
$(A \subseteq B) \wedge (B \subset C)$	$A \subseteq B \subset C$
$x \leq y$	$y \geq x$
$x < y$	$y > x$
$(x < y) \wedge (y \leq z)$	$x < y \leq z$
$w + (-x) + y + (-z)$	$w - x + y - z$

Table 1: Examples of alternate style options. The left side best reflects the internal representation. Variables are arbitrary just for examples. Line wrapping and choices regarding parentheses should also by done as style options (though not all of these are supported yet).

Two expressions that are the same apart from their style are deemed to be the *same* expression in **Prove-It**. The python == operation will return `True` when applied to two expressions that are the same up to their style settings. They are not the same object in every respect, of course, if they have different style settings. But they are the same with respect to proof derivations which is the entire purpose of **Prove-It**.

Style options may be used as an alternative to defining new expression classes and associated axioms for defining new notation. Table 1 lists many examples. Whether something should be done as a style option or as a separate expression type is not always obvious. There are trade-offs that must be balanced when deciding whether to define notation using style or a new expression class. Regarding $a < b \leq c < d$ as a shorthand notation for $(a < b) \wedge (b \leq c) \wedge (c < d)$ is very sensible; it simplifies the code and proofs, and is relatively straightforward for the user. As a counterexample, it is very tempting to regard $x \neq y$ as shorthand for $\neg(x = y)$. However, by defining a separate `NotEquals` class, we are able to write convenient and accessible methods for manipulating this specific kind of operation independent of the negation operation methods.

B.2 Parameter labels as a style choice

As mentioned in Appendix A in the definition of the **Lambda** expression class, **Lambda** expressions that are equivalent up to a relabeling of the parameters are deemed by **Prove-It** to be the *same* expression. While the machinery for this is different than the machinery for setting style options described above, it is essentially a style feature. Changing **Lambda** parameters does not alter the meaning of the expression, it only changes the way in which it is formatted. In lambda calculus terminology, this is known as alpha conversion.

Prove-It implements this feature by generating a canonical form whenever a **Lambda** expression is created. This is in the same spirit as De Bruijn notation used in lambda calculus though it does work differently (De Bruijn notation may assign different indices to different occurrences of the same variable, while we assign a consistent label for all occurrences of a variable in the same scope). Essentially, our canonical form assigns parameters to the first unused dummy variable going from $\_a$ to $\_z$ and continuing $\_aa$ to $\_az$, $\_ba$ to $\_bz$, etc. (it is unlikely so many variables would be needed, but there are endless possible dummy variables). For example, the canonical version of
$x \mapsto \left[\exists_y \ ((x + y + z) = 0)\right]$
is
$\_b \mapsto \left[\exists_{\_a} \ ((\_b + \_a + z) = 0)\right]$.

When there are a range of parameters, the canonical form is typically established in a similar manner. The canonical form of
$(x_1, \ldots, x_n, y_1, \ldots, y_n) \mapsto ((x_1 \cdot y_1) + \ldots + (x_n \cdot y_n))$
is
$(\_c_1, \ldots, \_c_n, \_b_1, \ldots, \_b_n) \mapsto$
$((\_c_1 \cdot \_b_1) + \ldots + (\_c_n \cdot \_b_n))$.
In this style it is not obvious why $\_a$ is skipped, but if we change the style option of the **ExprRange** within the `Add` **Operation** so that the `parameterization` is `explicit`, we have
$(\_c_1, \ldots, \_c_n, \_b_1, \ldots, \_b_n) \mapsto$
$\left((\_c_1 \cdot \_b_1) + .. \left(\_c_{\_a} \cdot \_b_{\_a}\right) .. + (\_c_n \cdot \_b_n)\right)$
where we now see that $\_a$ is being used to parameterize the **ExprRange** that runs from 1 to n.

In the above example, there is no question that all of the utilized indices of x and y were covered by the parameters, $(x_1, \ldots, x_n, y_1, \ldots, y_n)$. Therefore, we are free to relabel x and y. However, for expressions where there is any ambiguity (or is not sufficiently obvious) whether the parameters cover all utilized indices, relabeling for those variables is disabled. For example, x and y may not be relabeled in
$(x_1, \ldots, x_n, y_1, \ldots, y_n) \mapsto ((x_1 \cdot y_1) + \ldots + (x_m \cdot y_m))$
and therefore x and y remain as the labels in the canonical form. This expression alone gives no indication as to whether or not $m > n$. If $m > n$ in this example, there would be indices of x and y that are not covered by the ranges of parameters, and therefore there are free indexed variable occurrences (e.g., x_{n+1} and y_{n+1} are free). It would therefore be improper to relabel x and y. Even in instances where the coverage may seem obvious, as in
$(x_1, \ldots, x_n) \mapsto (x_1 + \ldots + x_{n-1} + x_n)$
Prove-It may error on the side of caution (and simplicity) and still regard x as non-relabel-able. The **Instantiation** inference rule [Appendix C.5] offers more versatility and can

effect relabeling in instances where canonical relabeling is not allowed. As an actual derivation step, it has more versatility than is allowable by a mere style feature (a derivation step can have dependencies to satisfy requirements while the justification of a style feature must be intrinsic to the expression itself).

C INFERENCE RULES

In this section, we describe each of **Prove-It**'s small set of inference rules used to construct derivation steps of a proof. We will use the standard notation for inferences rules with a horizontal line separating premises above the line from a conclusion below the line:

$$\frac{\text{Premises}}{\text{Conclusion}}$$

The premises represent any number of judgments and the conclusion represents a single judgement that may be derived from the premises. As we use this inference rule notation, we may either present explicit judgment assumptions within curly braces, or we will represent assumption sets in a set theoretic manner (e.g., $S \cup T \vdash X$ has, as assumptions the set union of S and T). When variable dependencies are relevant, the variable dependence will be explicitly shown. For example, we will use $P(x)$ to denote an expression in which x may (or may not) appear as a free variable, and we use $T(x)$ to denote a set of assumptions for which x may (or may not) appear as a free variable in each assumption.

C.1 Proof by assumption

In **Prove-It**, any expression may be proven to be true by assumption. That is,

$$\frac{}{\{A\} \vdash A} \quad \text{(ASSUMPTION)}$$

may be introduced as a valid judgment in a proof for any A. It does not matter if A is an expression that would normally have a Boolean type, and one must never assume that it has an intrinsic Boolean type just because it may be assumed. By assuming it, it is true (and therefore Boolean) *under that assumption*. To emphasize this point, note that we may prove, in **Prove-It**, that
$\{3\} \vdash 3$
which should simply be taken to mean that
$\{3 = \top\} \vdash 3 = \top$
as was discussed in Sec. 2. Of course, this assumption is bogus, so whatever conclusion one deduces under this assumption is worthless (but correct by **Prove-It**'s logic).

One may also have a range of assumptions using an **ExprRange**. Using a range of assumption, we can prove that the conjunction of these expressions is true. Thus,

$$\frac{}{\{A_1, \ldots, A_m\} \vdash A_1 \wedge \ldots \wedge A_m} \quad \text{(ASSUMPTIONS)}$$

Notice that we have no assumption here about m being a natural number. However, for this to be a valid assumption, m must be a natural number. But, again, we are not concerned about what we are able to prove under invalid assumptions. Introducing a logical conjunction here is important. It would never be valid (under valid assumptions) to prove that a tuple of expressions is true. Rather, it's the conjunction of these expressions, in this case, that forms the statement we are claiming to be true. In a sense, there is an implicit conjunction as a connective joining all assumptions. But it is only when a range of assumptions appears on the right of the turnstile that we need to make it explicit.

C.2 Axiom/theorem/conjecture invocation

Another way to introduce a judgment into a proof is by invoking an axiom, theorem, or even a conjecture. An axiom is regarded to be true as an asserted fact. Theorems are as valid as the axioms used, directly or indirectly, to prove them. A conjecture in **Prove-It** may be a well-known fact, but its conjecture status indicates that it has not be fully proven in the **Prove-It** system; it either has no proof, or its proof relies, itself, on conjectures. Within a given proof, however, these are accepted as valid judgments without question. The dependencies of a proof, discussed in Sec. 4.2.1, provide the additional information about what axioms were required by the proof as well as what unproven conjectures were relied upon.

C.3 Modus ponens

We can express our **Modus Ponens** inference rule, which is a generalization of standard modus ponens as

$$\frac{S \vdash A \qquad T \vdash A \Rightarrow B}{S \cup T \vdash B} \quad \text{(MP)}$$

This reduces to the standard modus ponens rule when S and T are both the empty sets. Modus ponens allows one to derive the consequent of an implication by proving its antecedent.

C.4 Deduction

The inference rule of **Deduction** is:

$$\frac{S \cup \{A\} \vdash B}{S \vdash A \Rightarrow B} \tag{D}$$

which is essentially the inverse of modus ponens. Modus ponens is used to eliminate an implication, deduction is used to introduce an implication. As a consequence of our **Modus Ponens** and **Deduction** rules and the fact that $\{A\} \vdash B$ in **Prove-It** means the same as $\{A = \top\} \vdash B = \top$, as discussed in Sec. 2, $A \Rightarrow B$ means the same as $(A = \top) \Rightarrow (B = \top)$ in the **Prove-It** system.

C.5 Instantiation

The **Instantiation** rule eliminates universal quantifiers, instantiating their variables to *any* expression. The following expresses the **Instantiation** rule when eliminating one universal quantifier for one variable:

$$\frac{S \vdash \forall_{x \mid Q(x)}\, P(x) \qquad T \vdash Q(Y)}{S \cup T \vdash P(Y)}$$

where x is a variable that may (or may not) occur free in $Q(x)$ and $P(x)$, and Y is *any* expression and serves as the replacement for x. There are specific reduction rules that dictate precisely how these replacements are to be performed that are described in Appendix D. In particular, this is implemented by effectively creating an ad-hoc lambda map for $x \mapsto Q(x)$ and $x \mapsto P(x)$ and applying these each to Y (performing beta reduction as it is called in lambda calculus terminology) to produce $Q(Y)$ and $P(Y)$, possibly performing other reductions in the process. These reduction rules may have requirements of their own which will be effectively added to the premises of the inference rule with their own assumption sets that must join the assumption set of the new judgment. Furthermore, if $Q(x)$ is a conjunction of conditions, such as $Q_1(x) \wedge Q_2(x) \wedge Q_3(x)$, these will effectively be individual premises. That is,

$$\frac{\begin{array}{l} S \vdash \forall_{x \mid Q_1(x), Q_2(x), Q_3(x)}\, P(x) \\ T_1 \vdash Q_1(Y) \qquad T_2 \vdash Q_2(Y) \qquad T_3 \vdash Q_3(Y) \end{array}}{S \cup T_1 \cup T_2 \cup T_3 \vdash P(Y)}$$

Although $Q_1(x), Q_2(x), Q_3(x)$ show up as a comma delimited list of conditions of the universal quantifier, this is merely a style choice for depicting a conjunction of conditions.

More generally, we may instantiate any number of variables:

$$\frac{\begin{array}{l} S \vdash \forall_{a,b,\ldots,z \mid Q(a,b,\ldots,z)}\, P(a, b, \ldots, z) \\ T \vdash Q(A, B, \ldots, Z) \end{array}}{S \cup T \vdash P(A, B, \ldots, Z)} \tag{INST}$$

where $a, b, \ldots, z$ is used to represent one or more variables which may include ranges of indexed variables and ranges of ranges of indexed variables, etc. via the **ExprRange** and **IndexedVar** expression types. $A, B, \ldots, Z$ are their respective replacements. Again, there are the caveats that a premise will be required for each *entry* of a conjunction of conditions and there may be additional premises from requirements of the lambda applications and related reductions resulting from replacing $a, b, \ldots, z$ with $A, B, \ldots, Z$ to produce $P(A, B, \ldots, Z)$ and $Q(A, B, \ldots, Z)$.

Prove-It's **Instantiation** rule may be used to eliminate any number of nested universal quantifiers for convenience as well as for the sake of brevity in the proof DAGs. The rule is essentially the same as above if you treat the nested universal quantifiers as a single universal quantifier over the union of the parameters and joining all of the conditions into one flattened conjunction. One may also regard this as multiple applications of the rule described here, but some relabeling of parameters along the way may be necessary for capture-avoidance purposes (discussed in Appendix D.2).

C.6 Generalization

The **Generalization** rule introduces an explicit universal quantification over any number of variables on the right side of the turnstile. There is essentially an implicit universal quantification over free variables of judgments acknowledging that a free variable appearing on both sides of the turnstile is the *same* variable. That means, if we are to add an explicit universal quantification over, say, x, on the right side of the turnstile, any assumptions involving x as free must be appropriately moved to the right side where it is now bound by the scope of the added quantifier. This is appropriately done by making such assumptions serve as *conditions* of the universal quantifier. A relatively simple form of the **Generalization** rule is

$$\frac{S \cup T(x) \vdash P(x)}{S \vdash \forall_{x \mid [\wedge](T(x))} P(x)}$$

where $[\wedge](T(x))$ is meant to denote a conjunction (in any order) over the assumptions, $T(x)$, that may (or may not) involve x as a free variable. This is similar to the ASSUMPTIONS rule where we needed to introduce a conjunction when performing proof-by-assumption on a range of assumptions. The assumptions of S must not have x as a free variable.

Adding extraneous conditions to the universal quantifier can only weaken the statement. We therefore allow, more

generally,

$$\frac{S \cup T(x) \vdash P(x)}{S \vdash \forall_{x \mid [\wedge](Q(x),T(x))} P(x)}$$

where $[\wedge](Q(x), T(x))$ is meant to denote a conjunction (in any order) over all of the assumptions, $T(x)$, that may (or may not) involve x as a free variable and some extra conditions, $Q(x)$ (which may or may not involve x) that only weakens the statement. More generally, we may add a universal quantifier for any number of variables:

$$\frac{S \cup T(a, b, \ldots, z) \vdash P(a, b, \ldots, z)}{S \vdash \forall_{a,\ldots,z \mid [\wedge](Q(a,\ldots,z),T(a,\ldots,z))} P(a, \ldots, z)} \quad \text{(GEN)}$$

where $a, b, \ldots, z$ (as well as $a, \ldots, z$ for brevity) is used to represent one or more variables which may include ranges of indexed variables and ranges of ranges of indexed variables, etc. via the **ExprRange** and **IndexedVar** expression types.

Prove-It's **Generalization** rule may be used to introduce any number of nested universal quantifiers for the sake of brevity in the proof DAGs, but one may simply regard this as multiple applications of the rule described here.

Much like **Modus Ponens** and **Deduction** are essentially inverses of each other for eliminating and introducing an implication, **Instantiation** and **Generalization** are essentially inverses of each other for eliminating and introducing a universal quantifier. As a consequence of our **Instantiation** and **Generalization** rules and the fact that $\{Q(x)\} \vdash P(x)$ is the same as $\{Q(x) = \top\} \vdash P(x) = \top$ in **Prove-It**, $\forall_{x \mid Q(x)} P(x)$ must mean the same as $\forall_{x \mid Q(x)=\top} P(x) = \top$ in the **Prove-It** system.

Literal Generalization (axiom elimination)

The essential difference between a **Variable** and a **Literal** is that a **Variable** is scoped within a particular judgment (and sometimes within a **Lambda** expression within the judgment) whereas a **Literal** has a universal scope (across judgments). Consider a judgment $\vdash P(\mathfrak{a})$ involving some **Literal**, $\mathfrak{a}$, that has been fully proven (to the level of axioms with no unproven conjectures). Now collect all of the required axioms for this proof that involve $\mathfrak{a}$. Say there are k of them denoted as $\vdash A_1(\mathfrak{a}), \ldots, \vdash A_k(\mathfrak{a})$. Then it must be the case that
$\{A_1(a), \ldots, A_k(a)\} \vdash P(a)$
using all of the axioms before *except* $\vdash A_1(\mathfrak{a}), \ldots, \vdash A_k(\mathfrak{a})$ (which are no longer necessary). We have replaced the **Literal** $\mathfrak{a}$ with the **Variable** a since we have brought all of the relevant facts pertaining to $\mathfrak{a}$ into the scope of a single judgment.

More generally, we can perform this transformation on any number of **Literal**s simultaneously. This is not a typical inference rule since the axiom requirements of the conclusion are reduced relative to the axiom requirements of the premise, but we shall denote this as

$$\frac{\begin{array}{l} S(\mathfrak{a}, \mathfrak{b} \ldots, \mathfrak{z}) \vdash P(\mathfrak{a}, \mathfrak{b} \ldots, \mathfrak{z}) \\ \text{via axioms } \mathcal{A}(\mathfrak{a}, \mathfrak{b} \ldots, \mathfrak{z}) \cup \mathcal{B} \end{array}}{\begin{array}{l} S(a, b \ldots, z) \cup \mathcal{A}(a, b \ldots, z) \vdash P(a, b \ldots, z) \\ \text{via axioms } \mathcal{B} \end{array}} \quad \text{(AXIOM\_ELIM)}$$

where $\mathfrak{a}, \mathfrak{b} \ldots, \mathfrak{z}$ are one or more **Literal**s which are respectively replaced with **Variable**s $a, b \ldots, z$. $\mathcal{A}(\mathfrak{a}, \mathfrak{b} \ldots, \mathfrak{z})$ denotes the set of axiom expressions (the right side of axiom judgments which are always assumption-less by our convention) of required axioms that involve any of the **Literal**s being converted, and $\mathcal{B}$ are the axiom expressions of required axioms that do not involve any of these **Literal**s.

With **Literal**s converted to **Variable**s and axioms converted to assumptions, we may now generalize these and/or any other **Variable**s using the GEN rule. Combining AXIOM_ELIM and GEN, we have a LITERAL_GEN rule as a single derivation step in **Prove-It**. For example, if we have fully proven some $\vdash P(\mathfrak{a})$ via axioms $\vdash A(\mathfrak{a})$ (and no other axioms involving $\mathfrak{a}$, we may derive
$\vdash \forall_{a \mid A(a)} P(a)$
in one LITERAL_GEN step, eliminating the $\vdash A(\mathfrak{a})$ as requirements in the process.

D REDUCTION RULES

As discussed in Appendix C.5, when the **Instantiation** inference rule is invoked, it creats ad-hoc **Lambda** expressions that define functions and applies those functions to operands determined by the desired replacements of the instantiation. The manifestation of the inference rule is dictated by these lambda application results. For example, if we instantiate
$\vdash \forall_{x,y \in \mathbb{N}} (x + y) \in \mathbb{N}$
with $x : 5$ and $y : b$ under the assumption that $b \in \mathbb{N}$, this results in
$\{b \in \mathbb{N}\} \vdash (5 + b) \in \mathbb{N}$
because $(x, y) \mapsto x \in \mathbb{N}$ applied to $(5, b)$ results in $5 \in \mathbb{N}$ (which is known to be true via axioms in the `proveit.numbers` theory package), $(x, y) \mapsto y \in \mathbb{N}$ applied to $(5, b)$ results in $b \in \mathbb{N}$ (which is true by assumption), and $(x, y) \mapsto (x + y) \in \mathbb{N}$ results in $(5 + b) \in \mathbb{N}$.

Lambda application is one reduction rule that **Prove-It** uses during **Instantiation**. Other reduction rules may be triggered in the process of performing a lambda application and making replacements of variables or ranges of variables as dictated by the lambda application. Reduction rules

are applied under a set of *assumptions* and may generate their own requirements. The assumptions come from the **Instantiation** invocation (i.e., the `instantiate` method of the `Judgment` class). The *requirements* will be effectively added to the premises of the inference rule of the instantiation and are allowed to use any of these assumptions. The actual assumptions of the concluding judgment of the effected **Instantiation** will be the union of the assumptions actually used by the premises. If an assumption is not utilized, it will be dropped.

D.1 Lambda application (beta reduction)

The **lambda application** reduction rule (beta reduction using lambda calculus terminology) needs to match operands to parameters, ensuring lengths match in the case of parameter ranges (parameters involving **ExprRange**s of **IndexedVar**s), to establish a mapping between parameters (or ranges of parameters) and their replacements. Then the replacements (substitutions) are performed, invoking other reduction rules in the process as appropriate.

As a simple example,
$(x, y, z) \mapsto ((x + y) \cdot z)$
applied to $(a + x, b \cdot y, b + y + x)$ yields
$((a + x) + (b \cdot y)) \cdot (b + y + x)$.

When there are ranges of parameters, it is a little more interesting. For example,
$(x_1, \ldots, x_n, y_1, \ldots, y_n) \mapsto ((x_1 \cdot y_1) + \ldots + (x_n \cdot y_n))$
applied to
$(1 \cdot 1, \ldots, n \cdot n, 1 + 1, \ldots, n + n)$
under the assumption $n \in \mathbb{N}$ yields
$((1 \cdot 1) \cdot (1 + 1)) + \ldots + ((n \cdot n) \cdot (n + n))$
with the requirements that
$\{n \in \mathbb{N}\} \vdash |\,(1 \cdot 1, \ldots, n \cdot n)\,| = |\,(1, \ldots, n)\,|$,
$\{n \in \mathbb{N}\} \vdash |\,(1 + 1, \ldots, n + n)\,| = |\,(1, \ldots, n)\,|$,
to ensure that the number of *operands* equals the corresponding number of *parameters*. Applying this same example lambda map to
$(0 \cdot 0, \ldots, k \cdot k, x, 1, \ldots, m, 0 + 0, \ldots, k + k, y, 1, \ldots, m)$
under the assumption that $k + 1 + m = n$ yields
$((0 \cdot 0) \cdot (0 + 0)) + \ldots + ((k \cdot k) \cdot (k + k)) + (x \cdot y) + (1 \cdot 1) + \ldots + (m \cdot m)$
with the requirements of
$\{k{+}1{+}m = n\} \vdash |\,(0 \cdot 0, \ldots, k \cdot k, x, 1, \ldots, m)\,| = |\,(1, \ldots, n)\,|$,
$\{k{+}1{+}m = n\} \vdash |\,(0 + 0, \ldots, k + k, y, 1, \ldots, m)\,| = |\,(1, \ldots, n)\,|$,
again ensuring that the number of *operands* equals the corresponding number of *parameters*.

A variable may occur in an expression in various forms, indexed over different ranges. In order to treat the various forms that a range of parameters may occur in an unambiguous and versatile manner, **Prove-It** allows one to specify "equivalent alternative expansions" for specifying various expansions for the different alternative forms. The rule in doing this is fairly simple and straightforward, but allows for a lot of versatility. Basically, if $x_i, ..., x_j$ is a range of parameters of the lambda expression, $(x_i, x_{i+1}, ..., x_{j-1}, x_j)$ could be an equivalent alternative expansion of $(x_i, ..., x_j)$ assuming $j - i \geq 1$ and have have its own replacement. These alternative expansions can provide the information needed to expand the variable in its various forms in **ExprRange** expansions described below. The requirements to allow for these alternative expansions are fairly straightforward. The tuple of indices of the equivalent alternative expansions must be equal to each other (e.g., $(i, i{+}1, ..., j{-}1, j) = (i, ..., j)$), and their replacements must be equal to each other. For example,
$(A_1, \ldots, A_m) \mapsto$
$\left(A_1 \wedge \ldots \wedge A_j \wedge \left[\forall_{A_i,\ldots,A_j}\ \left(A_i \vee \ldots \vee A_j\right)\right] \wedge A_m\right)$
applied to $(\neg B_1, \ldots, \neg B_{i-1}, \neg C_1, \ldots, \neg C_i, A \vee D)$
assuming $i \in \mathbb{N}^+$, $j = 2 \cdot i - 1$ and $m = j + 1$ and using $\left(A_1, \ldots, A_{i-1}, A_i, \ldots, A_j, A_m\right)$ and $\left(A_1, \ldots, A_j, A_m\right)$ as equivalent alternative expansions for $(A_1, \ldots, A_m)$ yields
$(\neg B_1) \wedge \ldots \wedge (\neg B_{i-1}) \wedge (\neg C_1) \wedge \ldots \wedge (\neg C_i)$
$\wedge \left[\forall_{A_i,\ldots,A_j}\ \left(A_i \vee \ldots \vee A_j\right)\right] \wedge (A \vee D)$
with the following requirements
$|\,(\neg B_1, \ldots, \neg B_{i-1}, \neg C_1, \ldots, \neg C_i, A \vee D)\,| = |\,(1, \ldots, m)\,|$,
$|\,(\neg B_1, \ldots, \neg B_{i-1})\,| = |\,(1, \ldots, i-1)\,|$,
$|\,(\neg C_1, \ldots, \neg C_i)\,| = |\,(i, \ldots, j)\,|$
$|\,(\neg B_1, \ldots, \neg B_{i-1}, \neg C_1, \ldots, \neg C_i)\,| = |\,(1, \ldots, j)\,|$,
$(1, \ldots, i-1, i, \ldots, j, m) = (1, \ldots, m)$,
$(1, \ldots, j, m) = (1, \ldots, m)$,
ensuring that the number of *operands* equals the corresponding number of *parameters* for the various ranges of indices of A (that is, for $A_1, \ldots, A_m$, $A_1, \ldots, A_{i-1}$, $A_i, \ldots, A_j$, and $A_1, \ldots, A_j$), and that the tuple of indices of the equivalent alternative expansions equal the original $(1, \ldots, m)$. This example is also interesting because the internal quantification over $A_i, \ldots, A_j$ is masking a portion of the $(A_1, \ldots, A_m)$ that is being mapped. Such masking is not advisable in practice, but it is an interesting test case.

D.2 Automatic relabeling (capture avoidance)

When replacements (substitutions) are made, we must ensure that the meaning of **Lambda** expressions are not altered by capturing variables which should be free. In lambda calculus terminology, this is known as capture avoidance. For example, given the expression

$a \mapsto a + b$
we cannot replace b with $f(a)$ unless we relabel the external a. The original expression represents a function that accepts a single argument and yields that argument plus something that is free. Replacing the free variable of this expression with something is not entirely free would change this meaning.

When such an invalid replacement is attempted in **Prove-It**, rather than giving an error, we simply relabel the conflicting bound variable to an available dummy variable that does not have a conflict. This is very much like the manner in which we choose dummy variables for making canonical forms of lambda expressions described in Appendix B.2. For the example above, replacing b with $f(a)$ would produce
$\_a \mapsto_{\_} a + f(a)$
where a and f are free but $\_a$ is bound. If there is a conflict and the variable is non-relabel-able for reasons discussed in Appendix B.2, there will be an error.

D.3 Operation replacement

D.3.1 Explicit operation replacement. When an operator is replaced with a **Lambda** expression, this lambda mapping is immediately applied as described in Appendix D.1. For example, $P(x, y)$ with
$P : [(a, b) \mapsto a + b]$
becomes $x + y$. This is not only convenient but necessary. Recall from Appendix A that an **Operation** restricts what is allowed as an *operator*. In particular, a **Lambda** expression is not allowed as an *operator*. But by immediately applying the lambda map (performing beta reduction), we avoid this issue and, importantly, we avoid Curry's paradox [Sec. 6.2].

The one other restriction is that the *body* of the **Lambda** that is replacing the operator *cannot* be an **ExprRange** type of expression. This is important to protect arity of operations, and lengths of tuples more generally. Without this restriction, you could generate a contradiction (there would be a paradox). For example, you can prove that
$\vdash \forall_f |(f(a), b, c)| = 3$
That is, the length of the tuple $(f(a), b, c)$ is 3 regardless of what f may be. But if you then instantiate f with
$f : x \mapsto x_1, \ldots, x_4$
where the right side of $\mapsto$ is an **ExprRange**, you *would* derive
$\vdash |(a_1, \ldots, a_4, b, c)| = 3$
But, $|(a_1, \ldots, a_4, b, c)|$ should be equal to 6. As long as $f(a)$ is not an **ExprRange**, though, we will not generate such a contradiction. So by disallowing the **Lambda** *body* to be an **ExprRange**, we avoid such a contradiction.

D.3.2 Implicit operation replacement. When an operator is replaced with a designated **Literal** operator of an **Operation** class, the operation is replaced by an instance of that derived class rather than the **Operation** class. For example, $P(x, y)$ with $P : +$ becomes $x + y$ with the correct `Add` type rather than a generic **Operation** type. This is not an important reduction rule for an independent proof checker which is only concerned about primitive expression types, but it is important for the proper behavior of **Prove-It** when generating a proof and formatting the resulting expressions.

D.4 ExprRange expansion

There are two kinds of reduction rules that are specific to **ExprRange** expressions that are applied in the process of performing a lambda application (beta reduction).

D.4.1 Indexed variable expansions. When the **ExprRange** contains one or more **IndexedVar** expressions with an index that is parameterized over the range, and the range of indexed variables is being replaced by an **ExprRange**, it may be reduced via either

Parameter dependent expansion is required whenever the parameter of the **ExprRange** occurs anywhere besides as an index of an indexed variable being expanded. In this case, the indices of the expanded indexed variable must match the original indices. For example,
$1 \cdot x_1 + \ldots + n \cdot x_n$ would be expanded, under
$(x_1, \ldots, x_n) : (a_1, \ldots, a_j, a_{j+1}, \ldots, a_n)$
to
$1 \cdot a_1 + \ldots + j \cdot a_j + (j + 1) \cdot a_{j+1} + \ldots + n \cdot a_n$
assuming $0 \leq j \leq n$ and requiring
$(1, \ldots, j, j + 1, \ldots n) = (1, \ldots, n)$.
However, an error would occur given
$(x_1, \ldots, x_n) : (a_1, \ldots, a_j, b_1, \ldots, b_k)$
as the expansion.

Parameter independent expansion is allowed otherwise. In this case, only the lengths of the expansion must match, not the indices themselves. For example, $x_1 \cdot y_1 + \ldots + x_n \cdot y_n$ could be expanded, under
$(x_1, \ldots, x_n) : (a_1, \ldots, a_j, b_1, \ldots, b_k)$
$(y_1, \ldots, y_n) : (c_1, \ldots, c_j, d_1, \ldots, d_k)$
to
$a_1 \cdot c_1 + \ldots + a_j \cdot c_j + b_1 \cdot d_1 + \ldots + b_k \cdot d_k$
assuming $j + k = n$.

D.4.2 Range reductions. When the **ExprRange** is known to be empty or contain only a single element, it may be reduced via either

Empty range reduction when the **ExprRange** is known to be empty it will be replaced with zero elements in a containing **ExprTuple**. For example, $(a, b_1, \ldots, b_n, c)$ with $n : 0$ reduces to (a, c).

Singular element range reduction when the **ExprRange** is known to contain only a single element it will be replaced with this one element. For example, $(a, b_1, \ldots, b_n, c)$ with $n : 1$ reduces to (a, b_1, c).

D.5 Conditional assumptions

The **Conditional** primitive expression type has one special reduction rule. Specifically, the condition of the **Conditional** may be introduced as an internal assumption in the process of performing a lambda application (or instantiation). Note that **Conditional**s are used implicitly in the conditions of quantifiers. For example, $\forall_{x \mid Q(x)} P(x)$ is internally represented as an operation with the $\forall$ operator applied to the lambda expression $x \mapsto \{P(x) \text{ if } Q(x) \text{ .}$, where $\{P(x) \text{ if } Q(x)$. is a **Conditional** expression, defined to be $P(x)$ when $Q(x)$ equals "true" but otherwise undefined for any practical purpose. Since the $P(x)$ value of the **Conditional** is only relevant when $Q(x)$ equals "true", **Prove-It** will add $Q(x)$ to the assumptions when making replacements of the $P(x)$ expression in the process of performing a lambda application (e.g., during an instantiation).

D.6 Automated equality reductions

Automated equality reduction is simply a way to replace an expression with another one that is provable equal to the original during an instantiation, typically to avoid notational inconveniences. For example, consider instantiating

$$\vdash \forall_{n \in \mathbb{N}^+} \left[\forall_{x_1, \ldots x_n, y_1, \ldots, y_n \mid (x_1 = y_1) \wedge \ldots \wedge (x_n = y_n)} \begin{array}{l} (x_1, \ldots, x_n) = \\ (y_1, \ldots, y_n) \end{array} \right]$$

with $n : 1$. Without the automated range reduction feature, you would end up with

$\vdash \forall_{x_1, y_1 \mid \wedge(x_1 = y_1)} (x_1) = (y_1)$

where $\wedge(x_1 = y_1)$ represents conjunction applied to a single operand $(x_1 = y_1)$. However, because we use automated range reduction for occurrences of unary conjunction, we instead obtain

$\vdash \forall_{x_1, y_1 \mid x_1 = y_1} (x_1) = (y_1)$

directly, but it does require

$\vdash \wedge(x_1 = y_1) = (x_1 = y_1)$

as a requirement to effect this automated equality reduction.

Automated equality reductions can be implemented for any expression class to perform reductions as desired. We prefer to limit its use, however, to instances where it avoids notational inconveniences (such as $\wedge(x_1 = y_1)$ which is notationally awkward). The danger of over-using this feature is that it can limit a user's control to derive the exact expressions they are trying to derive. For example, they may want to delay a reduction for pedagogical purposes. While the feature can be disabled for individual expression classes, over-use of the feature can still be inconvenient and confusing to a user. We therefore advise caution regarding the use of this feature.

When automated equality reductions are performed in an instantiation step, the requirements for performing these reductions (which are necessarily equality statements with the reducible expression on the left and the reduced version on the right) will be marked as "equality replacement requirements." This is useful information for an independent proof checker (automated or manual) which simply needs to replace reducible expressions with their reduced versions while effecting the instantiation step to confirm that it was implemented faithfully.

E BASIC THEORIES AND AXIOMS

In this section, we list some of the basic theory packages of **Prove-It** and their axioms. Ideally, axioms provide the most fundamental definitions of the various kinds of useful expressions that may be constructed.

Each axiom is a self-evident or asserted truth, tied to a particular theory package, with a meaning that is governed by its expression DAG. For simplicity, however, we only show a particular stylistic notation for each of these, with Table 2 as a legend of symbols for logic and set theory notation. The full expression DAGs are available on the **Prove-It** website [22].

These axioms are subject to change. In some cases, the best choice for the axioms is not obvious. They are chosen to be clear and fundamental, not necessarily minimalistic, and this is a subjective criteria.

It is intended that these basic axioms be consistent with standard, well-established mathematical norms. However, because we do not use intrinsic type restrictions or presume implicit domains of discourse, as discussed in Sec. 2, some of the axioms extend definitions beyond typical uses. For example, rather than intrinsically limiting logical negation to Boolean types, we only provide definitions for the Boolean types and furthermore assert that if a logical negation has a Boolean value then its operand must be a Boolean value (see axiom 4 of the `proveit.logic.boolean.negation` axioms below). This can allow us to conveniently infer the type of

an expression indirectly. We simply take an agnostic position with respect to the logical negation of a non-Boolean value; it is what it is and cannot be reduced. We have similar axioms for logical conjunction and disjunction that allow us to conveniently infer the type of their operands if we know that the conjunction/disjunction is a Boolean. Note, however, that we do not and can not, in our framework, similarly infer the type of an implication operand to be a Boolean. An implication is always Boolean in **Prove-It** regardless of the types of its operands. One should regard $A \Rightarrow B$ in **Prove-It** as no different from $(A = \top) \Rightarrow (B = \top)$, which is clearly either true or false. This is not a consequence of its axioms (the `proveit.logic.boolean.implication` axioms all have Boolean type restrictions). It is a consequence at a deeper core level of the lack of intrinsic type restrictions, the law of Deduction [see Appendix C.4], as well axiom 4 of `proveit.logic.booleans` axioms which allows us to deduce $\vdash A = \top$ from $\vdash A$.

- `proveit.core_expr_types.operations` axiom

(1) $\vdash \left[\begin{array}{c} \forall_{n\in\mathbb{N}}\forall_{f,x_1,\ldots,x_n,y_1,\ldots,y_n \mid (x_1,\ldots,x_n)=(y_1,\ldots,y_n)} \\ (f(x_1,\ldots,x_n) = f(y_1,\ldots,y_n)) \end{array}\right]$

- `proveit.core_expr_types.conditionals` axioms

(1) $\vdash \forall_a \ (\{a \text{ if } \top \ . = a)$

(2) $\vdash \forall_{a,Q} \left(\begin{array}{c} \{a \text{ if } Q \ . \\ = \{a \text{ if } Q = \top \ . \end{array}\right)$

(3) $\vdash \forall_{a,b,Q \mid Q\Rightarrow(a=b)} \left(\begin{array}{c} \{a \text{ if } Q \ . \\ = \{b \text{ if } Q \ . \end{array}\right)$

- `proveit.core_expr_types.lambda_maps` axiom

(1) $\vdash \left[\begin{array}{l} \forall_{i\in\mathbb{N}^+} \forall_{f,g} \\ \left(\begin{array}{c} \left[\forall_{a_1,\ldots,a_i} \left(\begin{array}{c} f(a_1,\ldots,a_i) = \\ g(a_1,\ldots,a_i) \end{array}\right)\right] \Rightarrow \\ \left(\begin{array}{c} [(b_1,\ldots,b_i) \mapsto f(b_1,\ldots,b_i)] = \\ [(c_1,\ldots,c_i) \mapsto g(c_1,\ldots,c_i)] \end{array}\right) \end{array}\right) \end{array}\right]$

- `proveit.core_expr_types.tuples` axioms

(1) $\vdash |()| = 0$

(2) $\vdash \forall_{i\in\mathbb{N}} \left[\forall_{a_1,\ldots,a_i,b} \ (|(a_1,\ldots,a_i,b)| = (i+1))\right]$

(3) $\vdash \left[\begin{array}{l} \forall_{i\in\mathbb{N}} \forall_{a_1,\ldots,a_i,b,c_1,\ldots,c_i,d} \\ \left(\begin{array}{c} \left(\begin{array}{c} (a_1,\ldots,a_i,b) = \\ (c_1,\ldots,c_i,d) \end{array}\right) = \\ \left(\left(\begin{array}{c} (a_1,\ldots,a_i) = \\ (c_1,\ldots,c_i) \end{array}\right) \wedge (b = d)\right) \end{array}\right) \end{array}\right]$

(4) $\vdash \forall_{f,i,j \mid (j+1)=i} \ ((f(i),\ldots,f(j)) = ())$

(5) $\vdash \forall_{f,i,j \mid |(f(i),\ldots,f(j))|\in\mathbb{N}} \left(\begin{array}{c} (f(i),\ldots,f(j+1)) = \\ (f(i),\ldots,f(j),f(j+1)) \end{array}\right)$

- `proveit.logic.booleans` axioms

(1) $\vdash \top$

(2) $\vdash \mathbb{B} = \{\top, \bot\}$

(3) $\vdash \bot \neq \top$

(4) $\vdash \forall_{A \mid A} \ (A = \top)$

(5) $\vdash \forall_{A \mid A=\top} A$

- `proveit.logic.booleans.implication` axioms

(1) $\vdash (\top \Rightarrow \bot) = \bot$

(2) $\vdash \forall_{A\in\mathbb{B} \mid (\neg A)\Rightarrow\bot} A$

(3) $\vdash \forall_{A\in\mathbb{B} \mid A\Rightarrow\bot} (\neg A)$

(4) $\vdash \forall_{A,B} \ ((A \Leftrightarrow B) = ((A \Rightarrow B) \wedge (B \Rightarrow A)))$

- `proveit.logic.booleans.negation` axioms

(1) $\vdash \neg\top = \bot$

(2) $\vdash \neg\bot = \top$

$\top$	true
$\bot$	false
$\mathbb{B}$	set of Boolean values (true or false)
$\Rightarrow$	implication
$\neg$	logical negation (not)
$\wedge$	logical conjunction (and)
$\vee$	logical disjunction (or)
$\forall$	universal quantification (forall)
$\exists$	existential quantification (exists)
$\nexists$	non-existence
$=$	equals
$\neq$	not equals
$\in$	set membership
$\notin$	set non-membership
$\cong$	set equivalence
$\ncong$	set non-equivalence
$\subset$	proper (strict) subset
$\subseteq$	subset
$\not\subset$	not proper subset
$\not\subseteq$	non-subset
$\supset$	proper (strict) superset
$\supseteq$	superset
$\not\supset$	not proper superset
$\not\supseteq$	non-superset
$\cup$	set union
$\cap$	set intersection
$\mathbb{N}$	natural numbers (whole number starting from zero)
$\mathbb{N}^{>0}$	natural numbers excluding zero
$\mathbb{Z}$	integers (positive and negative whole numbers)
$\mathbb{Q}$	rational numbers
$\mathbb{Q}^{>0}$, $\mathbb{Q}^{\geq 0}$, $\mathbb{Q}^{<0}$	positive, non-negative, and negative rational numbers (respectively)
$\mathbb{R}$	real numbers
$\mathbb{R}^{>0}$, $\mathbb{R}^{\geq 0}$, $\mathbb{R}^{<0}$	positive, non-negative, and negative real numbers (respectively)
$\mathbb{C}$	complex numbers

Table 2: Legend of symbols for logic and set theory notation and number sets.

(3) $\vdash \forall_{A \mid \neg A} \ (A = \bot)$
(4) $\vdash \forall_{A \mid \neg A \in \mathbb{B}} \ A \in \mathbb{B}$

- `proveit.logic.booleans.conjunction` axioms

(1) $\vdash (\top \wedge \top) = \top$
(2) $\vdash (\top \wedge \bot) = \bot$
(3) $\vdash (\bot \wedge \top) = \bot$
(4) $\vdash (\bot \wedge \bot) = \bot$
(5) $\vdash \forall_{A,B \mid (A \wedge B) \in \mathbb{B}} \ (A \in \mathbb{B})$
(6) $\vdash \forall_{A,B \mid (A \wedge B) \in \mathbb{B}} \ (B \in \mathbb{B})$
(7) $\vdash [\wedge] \ ()$
(8) $\vdash \forall_{m \in \mathbb{N}} \left[\forall_{A_1,\ldots,A_m,B} \left(\begin{array}{c} (A_1 \wedge \ldots \wedge A_m \wedge B) = \\ ((A_1 \wedge \ldots \wedge A_m) \wedge B) \end{array} \right) \right]$

- `proveit.logic.booleans.disjunction` axioms

(1) $\vdash (\top \vee \top) = \top$
(2) $\vdash (\top \vee \bot) = \top$
(3) $\vdash (\bot \vee \top) = \top$
(4) $\vdash (\bot \vee \bot) = \bot$
(5) $\vdash \forall_{A,B \mid (A \vee B) \in \mathbb{B}} \ (A \in \mathbb{B})$
(6) $\vdash \forall_{A,B \mid (A \vee B) \in \mathbb{B}} \ (B \in \mathbb{B})$
(7) $\vdash \neg \, [\vee] \ ()$
(8) $\vdash \forall_{m \in \mathbb{N}} \left[\forall_{A_1,\ldots,A_m,B} \left(\begin{array}{c} (A_1 \vee \ldots \vee A_m \vee B) = \\ ((A_1 \vee \ldots \vee A_m) \vee B) \end{array} \right) \right]$

- `proveit.logic.booleans.quantification.universality` axiom

(1) $\vdash \forall_{n \in \mathbb{N}^+} \ \left[\forall_P \ \left(\left[\forall_{x_1,\ldots,x_n} P(x_1,\ldots,x_n) \right] \in \mathbb{B} \right) \right]$

- `proveit.logic.booleans.quantification.existence` axioms

(1) $\vdash \forall_{n \in \mathbb{N}^+} \left[\forall_{P,Q} \left(\begin{array}{c} \left[\exists_{x_1,\ldots,x_n \mid Q(x_1,\ldots,x_n)} \ P(x_1,\ldots,x_n) \right] = \\ \neg \left[\begin{array}{c} \forall_{y_1,\ldots,y_n \mid Q(y_1,\ldots,y_n)} \\ (P(y_1,\ldots,y_n) \neq \top) \end{array} \right] \end{array} \right) \right]$
(2) $\vdash \forall_{n \in \mathbb{N}^+} \left[\forall_{P,Q} \left(\begin{array}{c} \left[\nexists_{x_1,\ldots,x_n \mid Q(x_1,\ldots,x_n)} \ P(x_1,\ldots,x_n) \right] = \\ \neg \left[\exists_{y_1,\ldots,y_n \mid Q(y_1,\ldots,y_n)} \ P(y_1,\ldots,y_n) \right] \end{array} \right) \right]$

- `proveit.logic.equality` axioms

(1) $\vdash \forall_{x,y} \ ((x = y) \in \mathbb{B})$
(2) $\vdash \forall_{x,y} \ ((y = x) = (x = y))$
(3) $\vdash \forall_{x,y,z \mid x=y, y=z} \ (x = z)$
(4) $\vdash \forall_{x,y} \ ((x \neq y) = (\neg \, (x = y)))$
(5) $\vdash \forall_x \ (x = x)$
(6) $\vdash \forall_{f,x,y \mid x=y} \ (f(x) = f(y))$

- `proveit.logic.sets.membership` axioms

(1) $\vdash \forall_{x,S} \ ((x \notin S) = (\neg \, (x \in S)))$

- `proveit.logic.sets.equivalence` axiom

(1) $\vdash \forall_{A,B} \ ((A \cong B) = [\forall_x \ ((x \in A) = (x \in B))])$
(2) $\vdash \forall_{A,B} \ \left(\left[A \ncong B \right] = \left(\neg \, (A \cong B) \right) \right)$

- `proveit.logic.sets.enumeration` axiom

(1) $\vdash \forall_{n \in \mathbb{N}} \left[\forall_{x,y_1,\ldots,y_n} \left(\begin{array}{c} (x \in \{y_1,\ldots,y_n\}) = \\ ((x = y_1) \vee \ldots \vee (x = y_n)) \end{array} \right) \right]$

- `proveit.logic.sets.inclusion` axioms

(1) $\vdash \forall_{A,B} \ ((A \subseteq B) = [\forall_{x \in A} \ (x \in B)])$
(2) $\vdash \forall_{A,B} \ \left(\left[A \nsubseteq B \right] = \left(\neg \, (A \subseteq B) \right) \right)$
(3) $\vdash \forall_{A,B} \ \left((A \subset B) = \left((A \subseteq B) \wedge \left(B \ncong A \right) \right) \right)$
(4) $\vdash \forall_{A,B} \ ((A \not\subset B) = (\neg \, (A \subset B)))$

- `proveit.logic.sets.unification` axioms

(1) $\vdash \forall_{m \in \mathbb{N}^+} \left[\forall_{x,A_1,\ldots,A_m} \left(\begin{array}{c} (x \in (A_1 \cup \ldots \cup A_m)) = \\ ((x \in A_1) \vee \ldots \vee (x \in A_m)) \end{array} \right) \right]$
(2) $\vdash \left[\begin{array}{l} \forall_{n \in \mathbb{N}^+} \forall_{S_1,\ldots,S_n,Q,R,x} \\ \left(\begin{array}{c} \left(x \in \left[\bigcup_{y_1 \in S_1,\ldots,y_n \in S_n \mid Q(y_1,\ldots,y_n)} R(y_1,\ldots,y_n) \right] \right) \\ = \left[\begin{array}{c} \exists_{y_1 \in S_1,\ldots,y_n \in S_n \mid Q(y_1,\ldots,y_n)} \\ (x \in R(y_1,\ldots,y_n)) \end{array} \right] \end{array} \right) \end{array} \right]$

- `proveit.logic.sets.intersection` axioms

(1) $\vdash \forall_{m \in \mathbb{N}^+} \left[\forall_{x,A_1,\ldots,A_m} \left(\begin{array}{c} (x \in (A_1 \cap \ldots \cap A_m)) = \\ ((x \in A_1) \wedge \ldots \wedge (x \in A_m)) \end{array} \right) \right]$
(2) $\vdash \left[\begin{array}{l} \forall_{n \in \mathbb{N}^+} \forall_{S_1,\ldots,S_n,Q,R,x \mid \exists_{y_1 \in S_1,\ldots,y_n \in S_n} \ Q(y_1,\ldots,y_n)} \\ \left(\begin{array}{c} \left(x \in \left[\bigcap_{y_1 \in S_1,\ldots,y_n \in S_n \mid Q(y_1,\ldots,y_n)} R(y_1,\ldots,y_n) \right] \right) \\ = \left[\begin{array}{c} \forall_{y_1 \in S_1,\ldots,y_n \in S_n \mid Q(y_1,\ldots,y_n)} \\ (x \in R(y_1,\ldots,y_n)) \end{array} \right] \end{array} \right) \end{array} \right]$

- `proveit.logic.sets.subtraction` axiom

(1) $\vdash \forall_{x,A,B} \ ((x \in (A - B)) = ((x \in A) \wedge (x \notin B)))$

- `proveit.logic.sets.comprehension` axiom

(1) $\vdash \left[\begin{array}{l} \forall_{n \in \mathbb{N}^+} \forall_{S_1,\ldots,S_n,Q,f,x} \\ \left(\begin{array}{c} \left(x \in \left\{ \begin{array}{c} f(y_1,\ldots,y_n) \ : \\ Q(y_1,\ldots,y_n) \end{array} \right\}_{y_1 \in S_1,\ldots,y_n \in S_n} \right) \\ = \left[\begin{array}{c} \exists_{y_1 \in S_1,\ldots,y_n \in S_n \mid Q(y_1,\ldots,y_n)} \\ (x = f(y_1,\ldots,y_n)) \end{array} \right] \end{array} \right) \end{array} \right]$

- `proveit.logic.sets.power_sets` axiom

(1) $\vdash \forall_{x,S} \ ((x \in \mathbb{P}(S)) = (x \subseteq S))$

- `proveit.logic.sets.cardinality` axioms

(1) $\vdash |\emptyset| = 0$
(2) $\vdash \forall_{x,S \mid |S| \in \mathbb{N}, x \notin S} \ (|S \cup \{x\}| = (|S| + 1))$
(3) Cardinality of infinite sets are not yet defined (requires ordinal numbers and additional axioms).

- `proveit.number.sets.naturals` axioms (1-5 are Peano's axioms)

(1) $\vdash 0 \in \mathbb{N}$
(2) $\vdash \forall_{n \in \mathbb{N}} \ ((n + 1) \in \mathbb{N})$
(3) $\vdash \forall_{m \in \mathbb{N}, n \in \mathbb{N} \mid (m+1) = (n+1)} \ (n = m)$
(4) $\vdash \forall_{n \in \mathbb{N}} \ ((n + 1) \neq 0)$
(5) $\vdash \forall_{S \mid S \subseteq \mathbb{N}} \ (((0 \in S) \wedge [\forall_{x \in S} \ ((x + 1) \in S)]) \Rightarrow (S \cong \mathbb{N}))$
(6) $\vdash \forall_x \ ((x \in \mathbb{N}) \in \mathbb{B})$
(7) $\vdash \{n \mid n > 0\}_{n \in \mathbb{N}}$

REFERENCES

[1] Leslie Lamport. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, 11(1):43–63, March 2012.

[2] D. Maslov, G. W. Dueck, D. M. Miller, and C. Negrevergne. Quantum circuit simplification and level compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):436–444, 2008.

[3] Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information*, 4(1), May 2018.

[4] Ebrahim Ardeshir-Larijani, Simon J Gay, and Rajagopal Nagarajan. Verification of concurrent quantum protocols by equivalence checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 500–514. Springer, 2014.

[5] Davide Venturelli, M. B. Do, Bryan O'Gorman, Jeremy Frank, E. Rieffel, Kyle E. C. Booth, T. N. Nguyen, P. Narayan, and Sasha Nanda. Quantum circuit compilation : An emerging application for automated reasoning. presented at ICAPS 2019 Workshop SPARK, 2019.

[6] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. Verified optimization in a quantum intermediate representation, 2019.

[7] Tao Liu, Yangjia Li, Shuling Wang, Mingsheng Ying, and Naijun Zhan. A theorem prover for quantum hoare logic and its applications, 2016.

[8] Lukas Burgholzer, Rudy Raymond, and Robert Wille. Verifying results of the ibm qiskit quantum circuit compilation flow, 2020. Available at https://arxiv.org/pdf/2009.02376.pdf.

[9] IBM Q. Ibm q. Last accessed on 9/30/2020 at https://www.research.ibm.com/ibm-q/.

[10] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2(79), 2018.

[11] Freek Wiedijk, editor. *The Seventeen Provers of the World*. Lecture Notes in Artificial Intelligence. Springer-Verlag Berlin Heidelberg, 2006. Foreword by Dana S. Scott. Book available at http://www.cs.ru.nl/~freek/comparison/comparison.pdf.

[12] Freek Wiedijk. The qed manifesto revisited. In R. Matuszwski and A. Zalewska, editors, *From Insight to Proof, Festschrift in Honour of Andrzej Trybulec*, pages 121–133. University of Bialystok, 2007. Available at https://www.cs.ru.nl/~freek/pubs/qed2.pdf.

[13] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information (10th Anniversary Edition)*. Cambridge University Press, New York, NY, 2010.

[14] Wayne Witzel, Mohan Sarovar, and Kenneth Rudinger. Versatile Formal Methods Applied to Quantum Information, 2015. prod.sandia.gov/techlib/access-control.cgi/2015/159617r.pdf.

[15] Robert S. Boyer, *et al.* The qed manifesto. In Alan Bundy, editor, *Automated Deduction – CADE 12*, pages 238–251. Springer Verlag, 1994. Available in Volume 814 of Lecture Notes in Artificial Intelligence and at http://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/wiedijk_2.pdf.

[16] Elliott Mendelson. *Introduction to Mathematical Logic (6th Ed.)*. Textbooks in Mathematics. CRC Press/Taylor & Francis Group, Boca Raton, Fla., 6th edition, 2015. Available at https://archive.org/details/IntroductionToMathematicalLogicByMendelson.

[17] Thomas Jech. *Set Theory (The Third Millennium Edition, revised and expanded)*. Springer Monographs in Mathematics. Springer-Verlag, Berlin, 2003.

[18] Francis Jeffry Pelletier. A brief history of natural deduction. *History and Philosophy of Logic*, 20:1–31, 1999.

[19] Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Acta Universitatis Stock-holmiensis, Stockholm studies in philosophy no. 3. Almqvist & Wiksell, Stockholm, G oteborg, Uppsala, 1965.

[20] Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Dover Publications, 2006.

[21] Per Martin-Löf. On the meaning of the logical constants and justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996. Available at http://docenti.lett.unisi.it/files/4/1/1/6/martinlof4.pdf.

[22] Prove-It, 2020. Last accessed on 9/24/2020 at www.pyproveit.org.

[23] Andrew David Irvine and Harry Deutsch. Russell's Paradox. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2016 edition, 2016.

[24] Kenneth Kunen. *Set Theory: An Introduction to Independence Proofs*, volume 102 of *Studies in Logic and the Foundation of Mathematics*. Elsevier, B. V., Amsterdam, The Netherlands, 1980.

[25] Jean E. Rubin. *Set theory for the mathematician*. Holden-Day series in mathematics. Holden-Day, San Francisco, 1967.

[26] Thomas Forster. *Logic, Induction and Sets*, volume 56 of *London Mathematical Society Student Texts*. Cambridge University Press, New York, NY, 2003.

[27] Stephen Cole Kleene and John Barkley Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 36(3):630–636, 1935.

[28] J. Barkley Rosser. Highlights of the history of the lambda-calculus. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, LFP '82, pages 216–225, New York, NY, USA, 1982. Association for Computing Machinery.

[29] Haskell B. Curry. The paradox of kleene and rosser. *Transactions of the American Mathematical Society*, 50:454–516, 1941.

[30] Haskell B. Curry. The inconsistency of certain formal logics. *The Journal of Symbolic Logic*, 7(3):115–117, 1942.

[31] Lionel Shapiro and Jc Beall. Curry's Paradox. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2018 edition, 2018. Available at https://plato.stanford.edu/archives/sum2018/entries/curry-paradox/.