Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Automated Theorem Proving in the Module System for Mathematical Theories

**Mark Hamilton · Florian Rabe · Michael Kohlhase ·
Gregg Zuckeman ·**

Received: date / Accepted: date

Abstract This thesis documents the development of a language-independent automated theorem prover for the Module System for Mathematical Theories (MMT) developed by the KWARC group.[4] This system aims to represent all of the logics and logical frameworks used in modern day mathematics and allow automated reasoning in all represented languages. The theorem prover leverages the Curry-Howard correspondence to store proofs, and represents languages in the logical framework LF enriched with dependent types. The theorem prover is built using an intelligent agent framework to maximize modularity and extendability. Some specific languages include propositional logic, first and higher-order logic, and physical theories with units [2] . This thesis will detail the architecture and the theory behind the prover as well as its performance.

Keywords MMT · Theorem Proving · Logics · Logical Independence · LF · ATP · Software Architecture

Mark Hamilton
Yale University, New Haven CT, 06520
E-mail: mark.hamilton@yale.edu

Florian Rabe
Jacobs University, Bremen Germany 28759
E-mail: florian.rabe@gmail.com

Michael Kohlhase
Jacobs University, Bremen Germany 28759
E-mail: m.kohlhase@jacobs-university.de

Gregg Zuckerman
Yale University, New Haven CT, 06520
E-mail: gregg.zuckerman@yale.edu

1 Introduction

At its heart, mathematics is the art of manipulating complex ideas by manipulating symbols. These symbols embody some of the greatest ideas in history, and have solved many of the worlds toughest problems. The power of mathematics lies in its formality, which provides a concrete answer to the questions of truth and provability. For thousands of years humans have manipulated mathematical symbols by hand to derive knowledge. However, as humans continue to research and develop new ideas, the body of mathematical and scientific knowledge grows exponentially. The current system results in knowledge across thousands of journals and publications. This makes it impossible for one person to learn more than a small fraction of the worlds mathematics in his/her lifetime. Thankfully, Maths formal and symbolic nature allows computers to aid in the work. Computers have the potential to organize, store, and contribute to our vast library of mathematical knowledge.

To illustrate this, one can think of mathematics as a language where one can express ideas about objects like numbers, sets and functions and prove them true or false. More formally, mathematical languages consist of symbols and a grammar to create well-formed sentences of the language. Each language of mathematics can then have multiple different theories. A theory of mathematics consists of axioms and proof rules. The axioms are well-formed sentences that are declared true and are the most basic units of mathematics. Ideally, they are so self-evident that they do not require any justification. (ex: an axiom from the theory of fields let x be an arbitrary number, 1*x = x). The proof rules allow the combination of axioms to create more true sentences. Together the axioms and proof rules give the symbols in the language their meaning by providing ways to manipulate them correctly.

The generalization of mathematics presented here is only the tip of the iceberg. Like physics, the foundation of mathematics has many different layers of generality and abstraction. The first layer of generality is the theory layer, which encompasses fields of mathematics like set theory and group theory. Each one of these theories uses the symbols and grammar of a language so one can abstract the language or logic layer. Next, one can abstract to a logical framework which provides a way to represent and reason about languages of mathematics. Finally, one can abstract to a module system which represents and reasons about logical frameworks. These top two layers are where the prover operates.

2 Background: Logics and Logical Frameworks

For decades, logic has been the foundation of all of mathematics. Most people have an intuitive understanding of logic, as it is pervasive in mathematics, computation, and natural language discourse. In this first chapter we will first develop logic formally. Through this, we will see that logics are actually well defined mathematical objects that can be reasoned about. To begin our introduction, it helps to consider one of the simplest and intuitive logics called Propositional Logic (PL). This is the logic of true and false propositions and boolean variables. This introduction is by no means thorough or completely rigorous, it is simply meant as an intuitive demonstration of some of the most important concepts in logic. For a more detailed introduction please see [5] or [1].

2.1 Logical Basics: PL

Mathematics often takes the form of manipulating strings of symbols on a page. However, these strings of symbols are very special, and generally mean something in the domain we are working in. If we

were to consider the set of all words made by mathematical symbols, most words would be completely nonsensical and not correspond to anything useful For example, we would not like to consider the word "$\Rightarrow\Rightarrow x \vee y =$". To narrow our focus to only the well formed statements of mathematics, we will consider a set of words formed by a mathematical object called a grammar that selects a structured subset of the set of all words in a language.

Propositional Logic is a mathematical object which formalizes reasoning about true and false statements of mathematics. We will begin with some definitions that characterize the syntax (written structure) of PL.

Definition 1 PL Signature
An alphabet that can be used to name variables, i.e. a set of symbols or characters Σ

Definition 2 PL Formulas
The formulas of PL are generated by the grammar below written in BackusNaur Form:

`form`	::=	*true*	*truth*
	|	*false*	*falsity*
	|	`form` $\wedge$ `form`	*conjunction*
	|	`form` $\vee$ `form`	*disjunction*
	|	`form` $\rightarrow$ `form`	*implication*
	|	$\neg$`form`	*negation*
	|	p where $p \in \Sigma$	*boolean variables*

These two definitions are used to create strings in the language of Propositional Logic, the math of truth and falsity. Notice that the definition contains some special logical characters like $\wedge, \vee$, etc, and an alphabet of variable characters. This definition carves out a much more sensible subset of the collection of all words over these symbols.

Now that we have formulas of mathematics, we must imbue them with meaning. First, we must select a subset of formulas that we consider irrefutably and intuitively true, called axioms. Second, we must write proof rules, or ways of transforming true formulas to get more true formulas. It is these rules that give the logical symbols with their meaning. In order to understand the proof rules in table 1, we must first understand the role of the sequent, $\vdash$. This symbol behaves in a similar way to the $\rightarrow$ symbol, however the former is meta-logical or used to define a logic whereas the latter is a symbol within the logic being defined. When a sequent proceeds a formula, it can be thought of as stating what is needed to make the formula derivable. In the case of an axiom like *true* which is always derivable we write $\vdash$ *true*. If a formula requires other formulas to be true, or mathematical objects to exist, we will list them as a collection of objects before the sequent. This is called the context and will generally be denoted as Γ

Proof rules or judgements are generally written vertically, and truth flows from above to below. One can think of them as mappings which take in arbitrary true statements of a certain type, and output true statements of a different type. Frequently, a logic can have several different collections of equivalent proof rules, or collections that can each prove the same formulas true and false. This paper will present the rules most often employed by human mathematicians called the Natural Deduction (ND) Calculus.

Fig 1 gives the introduction and elimination rules for ND as well as some structural rules. All rules are given for arbitrary Γ, A, and B.

As an example for how to read a proof rule, consider the first introduction rule, called "And ($\wedge$) Introduction". This rule states that given any formulas A and B which are both derivable given the

	Introduction	Elimination
$\wedge$	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\wedge I$	$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\wedge E_l \quad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\wedge E_r$
$\vee$	$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\vee I_l \quad \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\vee I_r$	$\dfrac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\vee E$
$\rightarrow$	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\rightarrow I$	$\dfrac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\rightarrow E$
$\neg$	$\dfrac{\Gamma, A \vdash \mathit{false}}{\Gamma \vdash \neg A}\neg I$	$\dfrac{\Gamma \vdash \neg A \quad \Gamma \vdash A \quad \Gamma \vdash C : \texttt{form}}{\Gamma \vdash C}\neg E$
true	$\dfrac{}{\Gamma \vdash \mathit{true}}\mathit{trueI}$	
false		$\dfrac{\Gamma \vdash \mathit{false} \quad \Gamma \vdash C : \texttt{form}}{\Gamma \vdash C}\mathit{falseE}$
	$\dfrac{A \in \Delta}{\Gamma; \Delta \vdash A}\mathit{Axiom} \quad \dfrac{}{\Gamma \vdash A \vee \neg A}\mathit{tnd}$	$\dfrac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash B}\mathit{cut}$

Fig. 1 Natural Deduction Proof Rules

context Γ, one can use the rule to conclude that the formula $A \wedge B$ is derivable in context Γ. Backwards reasoning refers to when the prover breaks up the target theorem by inverting the proof rules (going bottom to top) and attempting to get to axioms at every branch. In the above example if we are trying to prove $A \wedge B$, we know that it is only true if we can prove A and B so we can break apart the problem by going from the bottom of the judgement to the top to focus on those two formulas as sub-problems. The goal of forward reasoning is to work from axioms using the rules (top to bottom) to derive the target theorem. If we have already proven A and B we could combine those formulas using and introduction forward to get the proof.

2.2 Thinking About PL With Types

In the above section we have considered formulas created by the grammar in definition 2. Each one of these formulas share a common generating grammar, and are all examples of the same type of mathematical object, namely a formula. In type theory one labels each variety of mathematical object (or term) with a unique type denoted $Term : Type$, or more concretely $5 : \mathbb{N}$. Mappings can transform objects of one type to another, and are denoted with an arrow like $f(.) : \mathbb{N} \Rightarrow \mathbb{R}$. Note that the special symbols in the grammar like $\wedge, \vee, \rightarrow$ look a lot like binary mappings of type $\texttt{form} \times \texttt{form} \Rightarrow \texttt{form}$. More explicitly, all of these symbols can accept two formulas on either side, and yield another formula. This fact hints at a much deeper equivalence between computation and logic called the Curry-Howard

Correspondence, which provides an elegant way to interpret and computationally represent logics and proofs. For a more detailed discussion of the Curry-Howard correspondence please see [6]. In brief, the isomorphism states that there is an exact correspondence between PL and lambda calculus, a mathematical model for computation. In this correspondence, formulas or theorems become algebraic data types, and proofs are data of that data-type (also called terms). In this framework, Axioms become the first data types to be inhabited by data or proof. Proof rules are mappings between data types, and help move the data initially found at the axioms to other datatypes, proving other theorems. A theorem is true if its type is inhabited by data, false if it is uninhabitable.

2.3 Representing Logics with Logical Frameworks

Now that we have seen some basic mechanics of the logic PL, let us see how we can represent such a logic with the logical framework LF. LF is a tool to write down grammars, just as we did in definition 2. In LF's syntax any non-terminal symbol in a grammar becomes a type and any production is written as an n-ary map. In LF, the grammar for PL becomes:

$$\begin{array}{lll}
\texttt{form} & : & \texttt{type}. \\
\texttt{truth} & : & \texttt{form}. \\
\texttt{falsity} & : & \texttt{form}. \\
\texttt{conj} & : & \texttt{form} \to \texttt{form} \to \texttt{form}. \\
\texttt{disj} & : & \texttt{form} \to \texttt{form} \to \texttt{form}. \\
\texttt{impl} & : & \texttt{form} \to \texttt{form} \to \texttt{form}. \\
\texttt{neg} & : & \texttt{form} \to \texttt{form}. \\
V_p & : & \texttt{form}. \qquad\qquad \text{for } p \in \Sigma
\end{array}$$

Note that the notion of chaining arrows together to represent n-ary maps, as in the `conj`, is called currying. To represent variables that can be bound as in first order logic, LF uses square brackets to denote typed substitutions also called holes. This allows us to use LF in a manner identical to lambda calculus to define logical computations. For example, we can use LF to define the equivalence operator as:

$$\texttt{equiv} : \texttt{form} \to \texttt{form} \to \texttt{form} \;=\; [a : \texttt{form}][b : \texttt{form}](\texttt{conj}\ (\texttt{impl}\ a\ b)\ (\texttt{impl}\ b\ a)).$$

We can now write

$$\texttt{equiv}\ \texttt{true}\ \texttt{false}$$

and substitution yields

$$\texttt{equiv}\ \texttt{true}\ \texttt{false} \;=\; \texttt{conj}\ (\texttt{impl}\ \texttt{true}\ \texttt{false})\ (\texttt{impl}\ \texttt{false}\ \texttt{true})$$

If we remember the Curry Howard correspondence, theorems become datatypes and proofs become terms of that type. Proof rules are on a higher level, these rules take in proofs of theorems of **any** type and return proofs of a theorem related to the input theorem. This means that the return type of a proof rule depends on the input proof type. To represent this phenomenon, LF has a feature called dependent types, where the type of an object can be indexed by the terms of a given type. In other words, LF allows types that depend on terms. In LF, this is handled similarly to substitutions, but on the level of types where type substitution is denoted by curly braces instead of square brackets. For example, the proof rule "And Introduction" would be written as such:

$$
\begin{array}{lll}
\texttt{form} & : & \texttt{type}. \\
\texttt{proof} & : & \texttt{form} \rightarrow \texttt{type}. \\
\wedge I & : & \{F : \texttt{form}\}\{G : \texttt{form}\}\,\texttt{proof}\ F \rightarrow \texttt{proof}\ G \rightarrow \texttt{proof}\ (F \wedge G).
\end{array}
$$

Note how the proof operator has type $\texttt{form} \rightarrow \texttt{type}$ which is the same as a function from formulas to types, illustrating the nature of a dependent type. The actual proof rule has two type level holes denoted by . This type substitution behaves in a manner identical to that of its term level counterpart. One can read "and introduction" as a rule that takes in two formulas and their respective proofs, and returns a proof of their conjunction. The proof of conjunction's type is $\texttt{proof}(F \wedge G)$ which is indeed a type because the proof operator maps a formula to its own type.

This is the true power of LF, it can represent a wide array of language features like formulas, grammars, variables, binding, and proofs. Using this framework, the researchers of the KWARC group have implemented a wide array of logicsthrough a project called LATIN. Because all of these logics are written in LF, any theorem prover that can understand LF can automatically understand these logics.

3 The Module System for Mathematical Theories (MMT)

MMT is one step above the logical framework. In the same way that the symbols and formulas of a logical framework allows one to represent a logic, the symbols and formulas of MMT allow one to represent a logical framework.[4] MMT and many of the tools developed in its ecosystem are built in Scala because of its rich functional type system, object oriented language design, and its overall flexibility and scalability. MMT makes several design choices to increase generality, such as having a syntax written entirely in unused characters, which allows it to represent systems using any symbols in standard character libraries.

MMT focuses on achieving foundational independence, or the ability to reason about all kinds of mathematical foundations, without relying on a particular foundation such as set theory or type theory. This system aims to represent all of the logics and logical frameworks used in modern day mathematics such as ZFC, category theory, first order logic, higher order logic, abstract algebraic theories, theories with physical units, music theories, and many more.[2]. MMT is designed to represent mathematical theories in the most modular way possible, and allows theories to be linked together and constructed with theory morphisms. Theory morphisms are semantics preserving maps between mathematical theories, and stitch all of math together in the category of logics. A familiar example of this is how one can view a ring as a combination of semi-group and an abelian group with a distributive axiom that links the two.

4 Theorem Proving Architecture

The prover is written in the module system developed by the KWARC group called the Module System for Mathematical Theories (MMT). The prover uses an intelligent agent framework to prove theorems in any language of mathematics. To accomplish this, it represents logics in a meta logic called LF. This logic allows one to create notions such as truth, falsity, formulas, proofs, existence, universality and many other familiar mathematical notions.

One can think of the prover as a room of mathematicians cooperating on a blackboard to solve a problem. The prover contains a data structure called a blackboard, where other data structures called

agents interact and add their thoughts. This architecture is informally visualized in figure 2 The prover schedules these agents so that their work does not conflict when they change the state of the blackboard.

Currently, many theorem provers are written as a single complex program with many different functions and ways of splitting control flow. As more features are added to a prover of this type, the code exponentially increases in complexity. This makes the code brittle, and often inhibits extensibility. An agent based framework is designed to prevent this complexity buildup by introducing a layer of abstraction and a paradigm of software design called agents. The framework allows one to add functionality to a program by adding agents to the system that perform tasks. This small constraint in development keeps a program modular, succinct, and dry. Furthermore, an agent-based framework allows the sharing of code between provers and the exporting of proofs to other more specialized theorem provers.

In this software architecture, agents are stateful objects that can enact changes on shared data. One can think of an agent as an intelligent being who can read and edit shared data. This allows one to abstract many data transformations into agents who are responsible for transforming pieces of data. By coordinating the actions of agents, this also provides a uniform way of handling parallel processes without corrupting the shared data. This framework lends itself nicely to auction theory, and affords parameters tunable by machine learning.

Code for the MMT can be found at: https://svn.kwarc.info/repos/MMT/doc/html/index.html

Code for the prover can be found in the "/src/mmt-leo" folder of the repository.

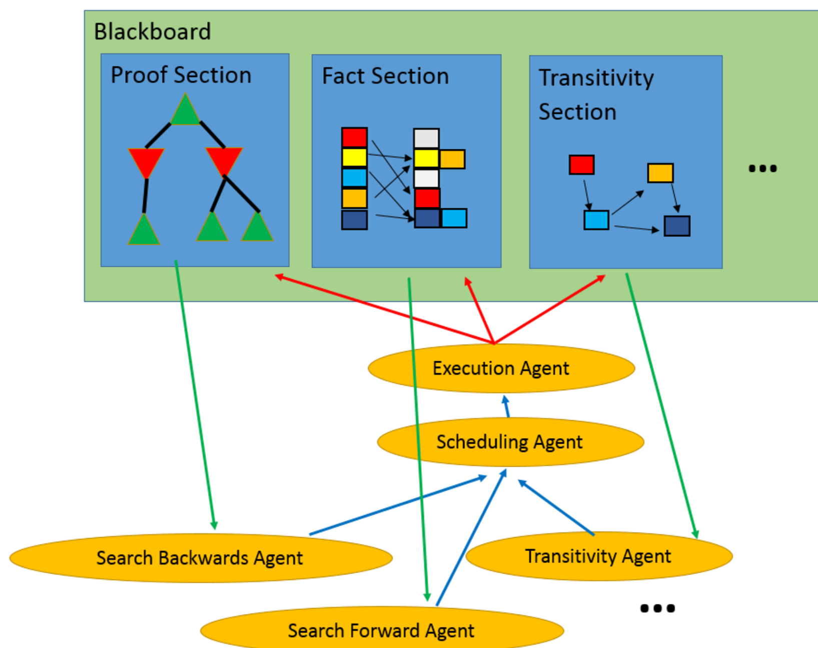

Fig. 2 Diagram showing the interactions of major classes in the prover. Green arrows from the sections to the agents represent reads, blue arrows between the agents represent message passing, red arrows from the execution agents to the sections represents writes and edits to the data held in the sections

4.1 Blackboard

This is the data-structure that holds and stores all mutable data in the theorem prover. All agents can view and edit data on the blackboard. This data storage area serves as a way to make progress on the problem in a way that is visible to all agents, so multiple tasks can be performed simultaneously. The blackboard contains several data-structures called sections, each section holds a specialized data-structure that represents a part of the proof state. The Section provides a standardizing layer that automatically adds change management/broadcasting and a uniform interface to access and edit many different types of data.

4.1.1 Names of Classes

Throughout this thesis I will refer to several types of objects, each one corresponding to a Scala class and each one serving its own role in the software. Below is a list the major classes and the intuitions behind them.

- Fact: A proven mathematical statement consisting of a $Term : Type$ where $Type$ represents the theorem and the $Term$ encodes the proof. These are stored in the Fact Database.
- Goal: A proven or unproven mathematical statement that forms a node of a proof tree. If the goal is proved it is called a closed goal, otherwise it is open. A goal has a context, formed by all of the assumptions required to get to that node in the proof tree. A goal has a type, and could have an associated term if it is proven. A goal is aware of its neighbors in the proof tree.
- Subgoals: Sets of goals that all need to be solved in order to solve the parent goal
- Alternatives: Sets of goals where only one goal needs to be solved to solve the parent goal
- Invertible Tactics: Methods of simplifying a goal during backwards reasoning in a way that does not loose information. One example of this in LF is Pi type introduction where dependent types are converted to implications and moved into the context. Tactics which do not preserve all information are called search tactics.
- Messages: Data passed between Agents in the prover
- Tasks: A specific kind of message which encodes a transformation of one of the mutable data-structures held in the blackboard
- Sections: Areas of the blackboard that hold the data-structures. These section wrappers provide uniform data access across all data-structures in the blackboard and handles change management and broadcasting to listening agents.

4.1.2 Proof Tree

This is the single most important piece of mutable data in the blackboard. This structure holds the state of the proof in a large tree where each node has an arbitrary number of children. As we have seen in section 2.1, a proof can be represented as a tree with a root labelled with the final theorem, leaf nodes labelled with axioms, and inner nodes labelled with rules of inference. Each node in a completed proof has a type or theorem, and a term or proof. The prover works by applying proof rules to reduce the theorems complexity and hopefully make it easier to solve. Often, this takes the form of splitting the problem or goal into sub-problems that can be solved and pieced together to solve the original problem.

For example, if we were trying to find a term/proof of the theorem $None : p \vee q$, we could try using the "Or Introduction" rule backwards to split the problem into two simpler alternatives: $None : p$ and

$None : q$. Now, if one of these alternative can be solved, the term/proof can be fed through the "Or Introduction" rule to yield a term/proof for the original theorem. In this method of proving called "Backwards Reasoning" we start with the final theorem and try to reduce to axioms. This process shows that it was built from solid building blocks.

The preceding paragraph demonstrates how a goal can be broken down into "alternatives", also called "Or-Nodes". These are collections of sub-problems where only one needs to be solved in order to solve the larger problem. Additionally, the prover could break a goal into "sub-goals" or sub-problems that all need to be solved to complete the larger problem. This often occurs when one branches the proof based on the truth or falsity of a certain fact. For example, if one trying to prove a fact about all numbers, it is often easier to consider the odd and even numbers separately. This allows one to split the problem into two simpler sub-problems. In each case, one has access to an additional fact, namely the parity of the number. Breaking up a problem into sub-goals increases the number of things to prove, but also makes the proofs "easier" as there is more information to work with. One should note that the prover completely expands a problem into its alternatives, or its subgoals. So a tree will alternate between sub-goals (need to solve all) and alternatives (need to solve 1) as passes from one layer of nodes to the next.

In code, this tree is represented with a Scala class for the nodes and fields connecting the node to its parent and list of children. This class has many utility functions for checking the location of a node, traversing, simplifying a goal, and pushing the proof terms through the labelled nodes. This data-structure is completely logic independent and operates on the module system level.

4.1.3 Fact Database

This section stores mathematical statements that have proofs, called facts. These facts can be transformed by proof rules to yield additional facts in a process called forward reasoning. For example, starting with two facts $Term1 : p$, $Term2 : q$, we can use the "And Introduction" rule going forward to generate the fact $AndIntroduction(Term1, Term2) : p \wedge q$. This fact can then be used like an axiom in many parts of the proof to close unproven goals of the proof tree. The fact database is initialized with the axioms of a given theory and is then enriched by feeding facts through proof rules.

After a few "enrichings" of the fact database, one finds that there are a very large number of facts available. To help manage the data, the data is hashed by the structure of the type of the theorem. More specifically, facts are hashed their abstract syntax trees, but truncated at a user-specified depth to limit the complexity of hash function calculation. Chaining is used to handle collisions. This data-structure allows one to query for facts based on the theorem they are intended to solve, which allows rapid closing of nodes in the proof tree if the fact exists in the database.

Many facts only become true under several assumptions. As a result, certain facts might only hold at a lower (more assumptions) state in the proof tree. As a result, all facts need to be labelled with their proof tree node or goal to allow for consistent reasoning.

4.1.4 Transitivity Database

The transitivity database is a semantics aware data-structure designed to respond to the existence of mathematical symbols that are transitive. Some examples of these symbols are $=, \leq, >, etc$ in the real numbers and $\Rightarrow, \leftrightarrow, \subset, etc$ in logic and set theory. More formally, a binary relation R is considered transitive if and only if xRy and yRz implies that xRz when written using infix notation. Often, the prover will discover many facts of shape xRy. These facts can link together, allowing proofs to flow

through the relations via the transitive axiom. For example, aRb, bRc, cRd, dRe implies that aRe if R is transitive. Naively, in order to discover these proofs with the standard fact database enriching process, it would require one to enrich the facts database n times where n is the number of times the transitivity axiom needs to be invoked. Humans tend to do these n steps automatically, but computers need a special data structure in order to reason quickly about transitive facts.

The theorem prover uses a directed graph with labelled edges and nodes to store these relations. Each node and edge is labelled with its corresponding fact that either proves the existence of the mathematical object, or the truth of the transitive statement. When the prover initializes, it scans the available set of proving rules for rules denoted with a "role" called either "transitive" or "equality". This role is an annotation made when declaring the logic, and is immediately recognized by the prover. When the prover sees this role, it adds a new graph based data-structure to the blackboard.

Each transitive relation gets its own directed graph, where nodes are labeled with facts and edges are labeled with proof that the relation holds between the two facts at the nodes. One can query this database for a proof of a fact $x\mathcal{R}y$, by asking to return a path between the x and y nodes in the graph. If a path exists, the database will automatically feed the proof terms through the transitivity axiom enough times to generate the correct proof.

Like the fact section, certain facts are only valid at lower nodes in the proof tree. As a result, all nodes and edges in the transitivity database need to be labelled with their corresponding goals at which they are valid. To help manage this information, only the most general possible facts are kept in the database. To maintain this invariant, fact nodes are merged when two nodes with the same type lie on the same branch of the proof tree. This ensures that the database can make the best use of available facts.

4.2 Agents

In addition to the mutable data-structures found in the blackboard, the prover also contains several agents which perform tasks, and mutate the data. Each agent can pass messages to and from other agents as well as read and edit certain data structures on the blackboard. Each agent encapsulates a certain behavior of the theorem prover and should be only weakly coupled to the other agents through message passing. In Scala, message passing is elegantly handled through case classes and pattern matching. [3]

4.2.1 Scheduling and Execution Agents

The scheduling agent handles the tasks generated by all other agents in the theorem prover. More specifically, at the end of each "Cycle" all other agents respond to changes in the blackboard by generating tasks that need to be run to change the state of the blackboard. The scheduling agent checks the read and write permissions of each task, and then bundles them into sets which can be executed in parallel without corrupting the data on the blackboard. This is currently done by sorting the tasks based on priority and excluding lower priority tasks that could corrupt data. Other types of scheduling agents could implement more sophisticated auction and utility based scheduling methods described in [7]. The scheduling agent then passes the task to the Execution Agent, which will run the tasks when the required computational resources become free.

4.2.2 Search Backward Agents

This agent will look at goals in the proof tree and attempt to break them into smaller sub-problems or alternatives. This process involves checking the goal against a list of LF specific backwards search tactics to see if any apply to the current goal. Once a tactic is chosen, a computational task is created which is sent to the scheduling agent. This task encapsulates the computations required to split a goal using a LF meta rule. Note that to extend the search depth, invertible backward rules are automatically applied whenever a new goal is added. As a result, this agent will apply any backwards rules that do not necessarily simplify the goal without losing information, these are called search backward rules.

4.2.3 Search Forward Agents

This agent will enrich the fact database by plugging known facts into the proof rules to create new facts. To eliminate cycles in the fact database, only a subset of the rules can be marked as forward rules using the MMT role "ForwardRule". This agent submits a task that has a low priority and is executed after the backwards search.

4.2.4 Transitivity Agent

This transitivity database is managed by a "transitivity agent" that is alerted whenever a fact or an open question is discovered with a syntax tree that matches that of a transitive, binary relation. The agent then either adds the fact to the data-structure, or queries the data-structure for the proof of the open question respectively. This has been found to improve performance on problems with a substantial amount of reasoning about equality and implication.

5 Theorem Proving in Theories with Units

We will now explore how to represent physical quantities and dimensional analysis within LF and MMT. The following section will present all code exactly as it is entered into MMT. The delimiters $\Box$ and $\star$ will be used as placeholders for MMT's non-printable character syntax. To create an MMT declaration, we must begin with a name, like *square_dim*. We then give this name a type using a colon, this corresponds to : `dim` $\rightarrow$ `dim`. Next, we can write the term by adding an expression with holes for substitutions (square brackets), ex: $\Box = [d]d * d$. One can then denote the notation for the operation with a #. To specify the notation, any characters can be used and the numbers $1, 2, 3...$ represent the arguments. The "prec" keyword allows one to specify binding strength. Higher precedences have higher binding strengths, which allows the omission of parentheses. Continuing with the example, one can express the notation for squaring a number as: $\Box\#1^2$ prec 10. Optionally, one can also provide a role for the theorem prover with the role keyword before ending the statement with a $\star$. The MMT syntax can take some getting used to, but its ability to define syntax allows one to write surprisingly readable statements despite the fact that one is representing a logic in LF which is in turn represented in MMT.

We will first begin our formalization of dimensional analysis by creating types that represent dimensions. this `dim` type will be the parent type for all of the specific dimensions like time, length, volume etc. We will also declare a scalar as a type of dimension, and we will define some algebraic operations that combine dimensions through multiplication and division. Note that once we define these operations and syntax, we can feel free to use the syntax elsewhere in the formalization. This allows our representation to be as close to natural human units as possible.

`real`	: `type`	$\square = num$	$\star$
`dim`	: `type`		$\star$
$scalar$	: `dim`		$\star$
$mult_dim$	: `dim` $\rightarrow$ `dim` $\rightarrow$ `dim`		$\square\#1 * 2\star$
inv_dim	: `dim` $\rightarrow$ `dim`		$\square\#1^{-}$ prec $10\star$
div_dim	: `dim` $\rightarrow$ `dim` $\rightarrow$ `dim`	$\square = [d, e]d * e^{-}$	$\square\#1/2\star$
$square_dim$	: `dim` $\rightarrow$ `dim`	$\square = [d]d * d$	$\square\#1^{2}$ prec $10\star$
$cube_dim$	: `dim` $\rightarrow$ `dim`	$\square = [d]d * d * d$	$\square\#1^{3}$ prec $10\star$

Next, we will flesh out the semantics of dimensions by adding proof rules. As detailed in section 2.3, we can use LF's dependent types to model proof rules. The dependent type syntax can be seen immediately in the curly brackets of type signature for the eq_dim rule. Note that scalar multiplication does not change the dimension, as it acts like an identity element in the theory of dimensions. We have also labelled the scalar multiplication rules with the Simplify role. This will allow the theorem prover to automatically perform these actions when simplifying a fact or goal.

eq_dim	: `dim` $\rightarrow$ `dim` $\rightarrow$ `type`	$\square\#1 = 2$ prec $-10\square role$ Eq$\star$
$assoc$	: $\{d :$ `dim`$, e :$ `dim`$, f :$ `dim`$\}d * (e * f) = (d * e) * f$	$\star$
$comm$	: $\{d :$ `dim`$, e :$ `dim`$\}d * e = e * d$	$\star$
$mult_scalar_right$	: $\{d\}d * scalar = d$	$\square role$ Simplify $\star$
$mult_scalar_left$	: $\{d\}scalar * d = d$	$\square role$ Simplify$\star$
$mult_inv_right$	: $\{d\}d * d^{-} = scalar$	$\square role$ Simplify$\star$
$mult_inv_left$	: $\{d :$ `dim`$\}d^{-} * d = scalar$	$\square role$ Simplify$\star$
inv_mult	: $\{d :$ `dim`$, e :$ `dim`$\}(d * e)^{-} = d^{-} * e^{-}$	$\square role$ Simplify$\star$
inv_inv	: $\{d\}d^{--} = d$	$\square role$ Simplify$\star$

As we continue with the formalization process, we need a way of representing a quantity of a given dimension. For this we will define a new dependent type called a quantity. This represents the type of objects like 10m or 1000s. Then we define a unit type which represents the type of the functions used to map real numbers e.g. 10 to quantities of a specified dimension e.g. 10m. The $makeQuantity$ function takes a unit and a real number and returns a quantity of the given dimension in the units provided. Finally algebraic operations are defined for manipulating these quantities.

`quantity`	: `dim` $\rightarrow$ `type`	$\square\#\$1\star$
`unit`	: `dim` $\rightarrow$ `type`	$\square = [d]$`real` $\rightarrow \$d\star$
$makeQuantity$	: $\{d\}$`real` $\rightarrow$ `unit` $d \rightarrow \$d$	$\square = [d, r, u]ur\square\#2'3$ prec $50\star$
$mult_quant$	: $\{d, e\}\$d \rightarrow \$e \rightarrow \$(d * e)$	$\square\#3 * *4\star$
div_quant	: $\{d, e\}\$d \rightarrow \$e \rightarrow \$(d/e)$	$\square\#3//4\star$
$mult_unit$	: $\{d, e\}$`unit` $d \rightarrow$ `unit` $e \rightarrow$ `unit` $(d * e)$	$\square\#3 * *4\star$ $= [d, e, u, v][r](r'u) * *(r'v)\star$
div_unit	: $\{d, e\}$`unit` $d \rightarrow$ `unit` $e \rightarrow$ `unit` (d/e)	$\square\#3//4\star$ $= [d, e, u, v][r](r'u)//(r'v)\star$
$mult_scalar$	: $\{d\}$`real` $\rightarrow \$d \rightarrow \$d$	$\square\#2@3\star$
$multiples_of$	: $\{d\}\$d \rightarrow$ `unit` d	$\square = [d, q][r]r@q\square\#multiples_of\ 2\star$

Next we will begin to specify some specific dimensions. In the actual implementation file, this is a separate mathematical theory which includes all of the previous definitions. This keeps the entire setup

modular and organized. This extra layer of organization is not shown here for ease of reading. Note that the syntactic sugars we have previously defined help legibility immensely.

$$
\begin{array}{lll}
length & : \texttt{dim} & \star \\
time & : \texttt{dim} & \star \\
mass & : \texttt{dim} & \star \\
temperature & : \texttt{dim} & \star \\
area & : \texttt{dim} & \square = length^2\star \\
volume & : \texttt{dim} & \square = length^3\star \\
velocity & : \texttt{dim} & \square = length/time\star \\
acceleration & : \texttt{dim} & \square = length * time^{-2}\star \\
force & : \texttt{dim} & \square = mass * acceleration\star \\
pressure & : \texttt{dim} & \square = force * length^{-2}\star
\end{array}
$$

We can also add prefixes to recover the standard terminology of milli, centi, and deci etc. These prefixes will act as maps from a unit type to the same unit type with a term formed by scaling the quantity by the given amount.

$$
\begin{array}{llll}
milli & : \{d\}\texttt{unit}\ d \to \texttt{unit}\ d & \square = [d,u][r](r/1000)'u & \square\#milli\ 2\star \\
centi & : \{d\}\texttt{unit}\ d \to \texttt{unit}\ d & \square = [d,u][r](r/100)'u & \square\#centi\ 2\star \\
deci & : \{d\}\texttt{unit}\ d \to \texttt{unit}\ d & \square = [d,u][r](r/10)'u & \square\#deci\ 2\star
\end{array}
$$

Now that we have some concrete dimensions at our disposal, we will create units for these dimensions. Just like the above definitions, these can be abstracted to their own mathematical theory that imports the dimensions above.

$$
\begin{array}{lll}
meter & : \$length & \star \\
second & : \$time & \star \\
gramm & : \$mass & \star \\
meters & : \texttt{unit}\ length & \square = multiples_of meter\star \\
seconds & : \texttt{unit}\ time & \square = multiples_of second\star \\
gramms & : \texttt{unit}\ mass & \square = multiples_of gramm\star \\
kelvin & : \$temperature & \star \\
kelvins & : \texttt{unit}\ temperature & \square = multiples_of kelvin\star \\
litre & : \$volume & \square = (1'decimeters) * *(1'decimeters) * *(1'decimeters)\star \\
litres & : \texttt{unit}\ volume & \square = multiples_of litre\star \\
tablespoon & : \$volume & \star \\
teaspoon & : \$volume & \star \\
cup & : \$volume & \star \\
tablespoons & : \texttt{unit}\ volume & \square = multiples_of(15'millilitres)\star \\
teaspoons & : \texttt{unit}\ volume & \square = multiples_of(5'millilitres)\star \\
cups & : \texttt{unit}\ volume & \square = multiples_of(250'millilitres)\star \\
minutes & : \texttt{unit}\ time & \square = multiples_of(60'seconds)\star \\
hours & : \texttt{unit}\ time & \square = multiples_of(60'minutes)\star \\
celsius & : \texttt{unit}\ temperature & \square = [r](r + 273 + (15/100))'kelvins\ \square\#1\ {}^\circ C\star \\
fahrenheit & : \texttt{unit}\ temperature & \square = [r](r - 32) * 5/9{}^\circ C\ \square\#1\ {}^\circ F\star
\end{array}
$$

We can now include the theory of physical units into any newly defined theory and can infer types and prove statements of mathematics with units automatically.

6 Results

The theorem prover is currently operational and can prove theorems in any language that can be represented in LF. The prover has been tested with a suite of around 80 proofs from propositional logic and first order logic, for a performance graph please see figure 3. The prover excels at shallow proofs because it performs breadth first search. The prover's depth is selectively deepened by the semantics aware transitivity database and the automatic expansion of goals with invertable backwards rules. However, the prover is not competitive with the state of the art in popular logics like PL and FOL. This is understandable given that the prover operates on a level of generality far beyond conventional provers which depend on language specific features. Furthermore, the prover has not undergone years of logic-dependent development. Nevertheless, it is one of the only theorem provers that can operate in the area of physical theories and theories with units. As more development resources are put into tuning and extending the provers understanding of semantics on a language-independent level, its skill in proving will increase across languages.

This section will demonstrate how to query the theorem prover, how it returns its proofs. The proofs are completely transferable and logically sound because they are type checked using the Curry-Howard correspondence. To call the prover into action, one must mark the term section of an MMT declaration with the underscore character. When MMT type checks the file, it will notice the missing term and try to infer it using the prover. Below is an example of three calls to the prover in the Propositional Logic with Natural Deduction calculus defined in 2.1.

$$\begin{array}{lll} \mathit{and_com} & : \{a,b\}\texttt{proof}(a \wedge b) \Rightarrow (b \wedge a) & \square = _ \star \\ \mathit{or_com} & : \{a,b\}\texttt{proof}(a \vee b) \Rightarrow (b \vee a) & \square = _ \star \\ \mathit{eq_com} & : \{a,b\}\texttt{proof}(a \Leftrightarrow b) \Rightarrow (b \Leftrightarrow a) & \square = _ \star \end{array}$$

The term or proof can be seen when one hovers over the named declaration when using MMT's jEdit plugin. One can also inspect the proof tree using the content viewer in the MMT jEdit plugin. This lets one view the proof in the natural tree structure with click-able and expandable nodes. Furthermore, the prover can create HTML outputs of the proof tree. The prover can also output the proof using the proof rule syntax declared in the logic that you are working in. For example the proof of the statement

$$A \Rightarrow A \wedge A$$

is written as

$$[A]\ \mathit{impI}\ [p]\ \mathit{andI}\ p\ p$$

which can be interpreted as a function that takes a formula A and its proof p and then feeds the proof through the and introduction rule called *andI* followed by the implication introduction rule *impI*. These two rules are functions that transform the proof of type A to the required proof of type $A \Rightarrow A \wedge A$. Because of MMT's binding and syntax capabilities this proof can be directly interpreted as a syntax tree instead of just a string.

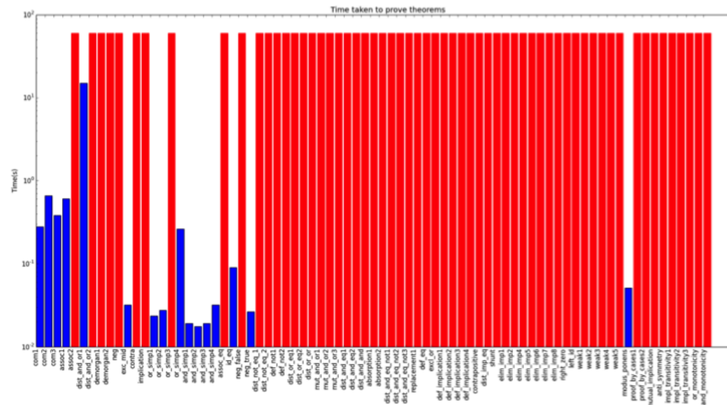

Fig. 3 Diagram showing the time taken to prove theorems from the test suite. Red bars means that the prover could not find a proof within 30 seconds. These proofs appear to be over 8 layers deep and need specialized, semantics aware search procedures

7 Conclusions and Future Work

This thesis has presented a logic independent theorem proving architecture which leverages the generality of MMT and the logical framework LF. This prover is operational and already has some automatic semantic recognition in the form of a transitivity database. The prover exemplifies the strengths of agent oriented design and will allow new researchers to easily extend the prover. The prover is currently one of the only provers in the literature able to prove theorems with physical units.

One future direction to pursue is the effect of machine learning proof techniques and parameters in a logic independent setting. One question that naturally arises is whether tuning the prover for one logic, could improve its performance for other logics as well. This would allow a kind of transfer learning to occur, and might offer hints at what kind of proving techniques are truly language independent.

Another future direction is to incorporate more general semantic recognition. One avenue is to try to find a leverage-able correspondence that allows one to derive a data-structure from a theorem, in a similar vein to how the transitivity database naturally corresponded to the transitivity axiom.

Currently, a Term database is also in development which will split the facts database by functional facts and object facts. This would allow the fact database to quickly generate new mathematical objects by only enriching the object facts. These object facts could support term based queries to increase the speed of finding witnesses or counterexamples to theorems.

Acknowledgements This paper would not be possible without the incredible help and support from Dr. Florian Rabe who supervised this project and provided me (Mark Hamilton) with many helpful resources, design considerations, and code for interfacing with MMT. Furthermore, I would like to acknowledge and thank Professor Michael Kohlhase for welcoming me into his lab to develop the theorem prover and Professor Gregg Zuckerman for advising this thesis and establishing the collaboration between Yale and Jacobs University.

References

1. Andrews, P.B.: An introduction to mathematical logic and type theory, vol. 27. Springer Science & Business Media (2002)
2. Codescu, M., Horozal, F., Kohlhase, M., Mossakowski, T., Rabe, F.: Project abstract: logic atlas and integrator (latin). In: Intelligent Computer Mathematics, pp. 289–291. Springer (2011)
3. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science **410**(2), 202–220 (2009)
4. Rabe, F.: The mmt api: a generic mkm system. In: Intelligent Computer Mathematics, pp. 339–343. Springer (2013)
5. Rabe, F.: Integrated lecture notes on logic (2014)
6. Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry-Howard isomorphism, vol. 149. Elsevier (2006)
7. Wisniewski, M., Steen, A., Benzmüller, C.: Leopard—a generic platform for the implementation of higher-order reasoners. arXiv preprint arXiv:1505.01629 (2015)