Linking HOL Light to Mathematica using OpenMath

Ons Seddiki

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science (Electrical & Computer Engineering)

at

Concordia University

Montréal, Québec, Canada

August 2014

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Ons Seddiki

Entitled: Linking HOL Light to Mathematica using OpenMath

and submitted in partial fulfilment of the requirements for the degree of

Master of Applied Science (Electrical & Computer Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final Examining Committee:

Dr. M. Z. Kabir
________________________ Chair
Chair's name

Dr. J. Bentahar
________________________ Examiner
Examiner's name

Dr. A. Hamou-Lhadj
________________________ Examiner
Examiner's name

Dr. S. Tahar
________________________ Supervisor
Supervisor's name

Approved by ____
Chair of Department or Graduate Program Director

__________ 2014 ____________________________________
Dean of Faculty

ABSTRACT

Linking HOL Light to Mathematica using OpenMath

Ons Seddiki

One of the most important benefits of using a theorem prover system is the absolute accuracy of the obtained result. However, solving mathematical problems often requires both deductive reasoning and algebraic computation. This issue is due to the fact that many real-life problems can be described with equations for which we cannot find easily symbolic (or closed-form) solutions and therefore we are not able to formalize them using the theorem prover. In other cases, some applications require well developed libraries and a deep knowledge of the theories to formalize simple expressions. A straightforward way to overcome these issues is the use of computer algebra systems or numerical approaches which are known to be the most efficient tools in symbolic computation. However, to preserve the soundness of the computation, the results of these systems should be formally verified. In this thesis, we present a general architecture to connect HOL Light, a higher-order logic theorem prover, to any mechanized mathematical system that supports the mathematical standard OpenMath. We implemented a prototype, called HolMatica, which links HOL Light to the computer algebra system Mathematica through OpenMath. We describe our implementation of a HOL Light translator which converts HOL Light statements into OpenMath object and vice-versa.

To My loving Parents

ACKNOWLEDGEMENTS

First and foremost, I would like to thank the almighty ALLAH.
Throughout these two years I had bad and good moments; however, at the end we usually forget about the bad and remember only the good ones and specially the best among them. For that, it would not be possible with having special persons in my life. So I would like to thank them through this little note which never can express my gratitude toward them.

I owe my deepest gratitude and thanks to the Tunisian Government and the University Mission of Tunisia in Montreal for their support that allowed me to carry out my studies in comfortable conditions.

Second, I sincerely thank my supervisor, Dr. Sofiène Tahar for giving me the opportunity to work on this project. He was very motivating, supportive and guided me efficiently throughout my Master's thesis. I have learned a lot from him with respect to research, academic and life in general. I consider him not only as my supervisor but also as my second father. I also would like to thank Dr. Cvetan Dunchev for his time and support. I express my heartfelt gratitude to him.

Third, I sincerely thank my colleagues at the Hardware Verification Group (HVG) at Concordia University for their timely suggestions during my research. Without their guidance, expert advice, support and continual encouragements, this thesis would not have been possible.

Then, my friends in Canada and in Tunisia gave me their constant support, love and encouragement, I can never thank them enough.

Last but not least, I thank my mom, dad, brother and Ahmed, for their constant support and their prayers. Their support was invaluable in completing this thesis. I could not do it without having them in my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

API	Application Programming Interface
CAS	Computer Algebra System
CD	Content Dictionary
GUI	Graphical User Interface
HOL	Higher-Order Logic
LK	Logistischer Klassischer
ML	Meta Language
MMS	Mechanized Mathematical Systems
OM	OpenMath
PA	Proof Assistant
PVS	Specification and Verification System
TCP/IP	Transmission Control Protocol/Internet Protocol
TPS	Theorem Prover System
XML	Extensible Markup Language

Functionality Description

Chapter 1

Introduction

1.1 Motivation

In the last decades, Theorem Prover Systems (TPSs) [34] have been used in the modeling and analyzing of many hardware and software systems. Recently, TPSs are employed in new areas such as optics, probabilistic and hybrid systems. The use of TPSs in analyzing complex systems is advantageous over conventional methods such as Computer Algebra Systems (CASs) [44] thanks to their inherent soundness and completeness. In addition, TPSs are more reliable and precise than CASs, but CASs are easier to use and more popular. However, in many cases the modeling and analysis of optical systems, for example, involves combining symbolic computation and logic reasoning.

The complementarity of TPSs and CASs lies in the fact that both of them are used to help people perform formal symbolic manipulation. In fact, CASs are mostly used to evaluate or compute complex mathematical expressions such as arithmetic computations, matrix operations or polynomial expressions. Similarly, TPSs are used to formally verify mathematical models driven by a system of logical inference rules with the respect to a given specification.

The integration of procedural algebraic knowledge and deductive knowledge is useful in many situations. The first situation is where the mathematical equations have non-closed-form solution. Often because of the physical nature of hardware systems, one cannot find a closed form solution for the system to be formalized in TPSs. Second, such an analysis requires the use of well developed libraries of the TPSs such as the simplification of complex mathematical expressions involving multivariate calculus.

The first problem can be addressed by using well-known numerical techniques which are readily integrated in computational tools such as MATLAB [15]. On the other hand, the latter can be addressed using CASs (e.g., Mathematica [12] and Maple [10]) which are considered to be most efficient tools for computing symbolic solutions automatically. Both of these techniques have some known limitations of incompleteness and unreliability in case of numerical techniques and CASs, respectively.

In this thesis, our main idea is to leverage upon the expressive nature and soundness of the higher-order logic theorem prover as much as possible, but at the same time we want to make use of the advantages of the CASs. In order to handle these situations, we propose to provide a bridge connecting the HOL Light theorem prover [26] with Mechanized Mathematical Systems [58] (MMSs) such as symbolic and numerical techniques based tools.

In this bridge, the given equation is first transferred to CAS in order to be simplified symbolically. If it is successfully simplified, then we transfer it to the theorem prover (ideally by first certifying the simplification) in order to pursue the proof process. In case the equation cannot be simplified symbolically, we have no option but to switch to numerical approaches.

In the following we will present two categories of MMSs which are TPS and CAS. Then, we will give an overview about the related work regarding the integration of symbolic computation and logic reasoning. We will describe also our proposed methodology linking HOL Light to Mathematica. In addition, we will outline the contributions of our work. Finally, we will conclude this chapter by presenting the outline of the rest of this thesis.

1.2 Mechanized Mathematical Systems

A mechanized mathematics system (MMS) is a computer system that supports and improves mathematical processing by applying deductive reasoning or performing symbolic/numeric computation [58]. There are presently two major types of MMSs: TPS and CAS.

1.2.1 Theorem Prover Systems

Theorem proving is a formal verification technique used to prove the correctness of a system. It consists of formalizing the system model and its properties (specifications) using mathematical logic [47]. Theorem proving is more efficient and provides a rich formalism compared to other verification methods such as model checking and equivalence checking [45]. Often a limitation of the model checking systems is the combinatorial blow up of the state-space which is known as the state explosion problem. Most real life problems have models which belong to the latter. However, a possible solution to this limitation is the use of a TPS.

A theorem prover system is a software tool which formally verifies any hardware or software system that can be described mathematically. A TPS proves that a system model satisfies its specification which is formalized in a logical system. The core of a TPS consists of a collection of basic theories and inference rules. The use of these formal notions help users to define and formalize any system that can be described theoretically. The most common deductive systems which are integrated in TPSs are the Natural Deduction and the Sequent LK Calculus. For example, the first calculus is used by TPSs such as HOL Light and Isabelle [7]. An extended version of the second calculus is used by KeYmaera [8]. Both calculus are isomorphic and sound and complete for the first-order logic. Since the propositional logic is decidable, TPSs for it are fully automated. However, first-order logic is semi-decidable and therefore the TPSs need a guidance through the proof steps, although there are some first-order theories which are decidable and for them there are effective procedures. Higher-order logic TPSs such as HOL Light are sound, but the logic is incomplete. The last limitation does not affect the verification process and therefore the interactive TPSs remain very strong tools for verifying properties of complex real life systems [47]. The soundness of the TPS is assured by the fact that each new theorem is derived from a small set of basic axioms and inference rules, and theorems that have been already proved. Moreover, the use of TPS requires a certain knowledge of its internal functions because the formalization is a difficult process. In fact, to prove a theorem already done by paper and pencil, one needs to add many steps which consist of choosing and applying the appropriate tactics to the list of assumptions. Besides, one needs not only to formalize the mathematical model of a hardware or software system or different mathematical theories but also its properties as well as the whole context that is needed to run the intended system [50, 49]. Examples of theorem proving systems are HOL [5] , HOL Light, Coq [25], Isabelle, PVS [23], etc.

1.2.2 Computer Algebra System

Computer Algebra System (CAS) is a software program for simplifying mathematical expressions. It is also known under other names such as symbolic calculations or analytic calculations [44]. Examples of CASs are Maple, Mathematica, Axiom [1], Reduce [24], etc. CASs are mostly used by mathematicians and scientists to evaluate mathematical expressions such as arithmetic manipulations, polynomials operation and matrix operations, for example the evaluation of eigenvalues for linear equations. Moreover, a CAS provides a user-friendly interface, a syntax which is similar to traditional manual computations and a fully automated system. For all these reasons, CASs are more popular and easier to use rather than using TPSs. In addition, CAS usually works faster than TPS since it is optimized for computing very complex mathematical expressions [49].

Similar to TPS, CAS provides formal symbolic mathematical manipulation. However, the result might not be accurate since the CAS evaluation does not take into consideration side conditions. For instance, given "x/x" to Mathematica, it returns 1, but "$x/x = 1$" holds only when "$x \neq 0$". This, however, will not hold in a TPS.

A special remark should be made regarding the soundness of the approach. Since a CAS does not implement any logical deductive system, its results must be verified by the theorem prover always when it is possible. Therefore, with the help of Mathematica, for example, we can reduce the process of finding a proof of a sub-goal which is in a sense non-deterministic, to a deterministic one. That means that we just have to verify the result, not to find it. The analogy with a deterministic Turning machine and a non-deterministic one, respectively, is straightforward. However, there are cases where the result of the CAS cannot be verified directly due to the lack of some theory, for example in cases of approximations. In these cases we might have some local inaccuracy, but still the results could be useful for analysis.

Solving mathematical problem often requires the application of both algebraic knowledge and deductive knowledge. In the following section, we will present the related work regarding the combining of the symbolic computation and the analytic reasoning.

1.3 Related Work

Verifying mathematical statements by TPS often requires algebraic computation. In the literature, many researchers have addressed the issue of combining symbolic/numeric computation with logical reasoning. In general, there are four approaches. One solution is building a CAS inside a TPS. The second one is building a TPS inside a CAS. The third approach implements a bridge between a TPS and a CAS. The fourth approach is to define a framework using a standard for mathematical information that can be exchanged between TPSs and CASs. In this section, we will give an overview of the state-of-the-art in connecting theorem prover and computer algebra.

1.3.1 CAS inside a TPS

Integrating a CAS inside a TPS is often required in situations where a user needs to calculate some operations such as the addition of fractions or the calculation of a derivative of a polynomial. In these cases, using TPS is non-trivial because it involves a good knowledge of the TPS libraries while using CAS is easier and simpler.

In [51], Kaliszyk *et al* proposed a user interface of a CAS that is built on top of HOL Light. This work is a prototype implemented in HOL Light. Its architecture is based on three independent parts. The first one is the user interface where a user can write the mathematical expression that needs to be simplified. The second part is the CAS conversion which is responsible of computing or evaluating the expression. The third part represents the knowledge to the specific CAS. This part is separate a form the system and contains methods that match the term to one rule among a list of the corresponding rewriting rules. In this work, HOL Light is used to prove the correctness of the simplified output result by CAS. Given an expression that needs simplification within HOL Light, the user writes it in the user interface. Then, the CAS conversion checks whether the given term matches with one of the rewrite rules existing in the knowledge base specific to the CAS which represents the set of theorems and conversion understandable by the CAS. Once the computation is complete, if the returned result is correct the CAS conversion returns a theorem which represents the certification of the input term. Otherwise, it returns an instance of a reflexivity theorem where the input is the same as the output. This prototype uses mathematical expressions with a precise semantic and according to this semantic HOL Light checks the soundness of the input statement. The advantage of this work is that all calculations done by HOL Light are certified by the architecture itself. However, as this system generates the results and their certifications, this application has lower performance than using traditional CAS in terms of execution time. In addition, the integrated CAS is used only for HOL Light and cannot be accommodated to other TPSs such as HOL or Isabelle.

In [8], Platzer *et al* proposed an automated and interactive theorem prover based on first-order logic for hybrid programs. KeYmaera combines deductive reasoning based on the theorem prover KeY [30] and algebraic algorithms using the computer algebra system Mathematica. It is fully automated and it includes also a user interface written in JAVA. Figure 1.1 describes the architecture of this hybrid theorem prover. It includes the proof rules bases that is used to prove the correctness of an expression. Moreover, the automatic proof strategies is used in order to simplify the proof steps of complex mathematical expressions.

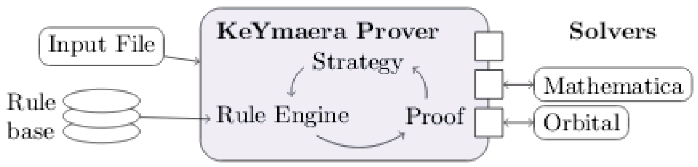


Figure 1.1: Architecture of the KeYmaera Prover [55]

1.3.2 TPS inside a CAS

This approach is to build a TPS inside CAS. This implementation represents usually an extension of the CAS to be able to prove the correctness of the simplified result. Theorema [36], for example, is a framework which uses advanced features of the kernel and the front end of Mathematica. Theorema allows users to prove, solve and compute mathematical expression. The use of Theorema is similar to the use of Mathematica. knowing the syntax of Theorema, the user has to define the mathematical expressions in the notebook (front end) of Mathematica in order to prove or compute them [37].

In [31], Clarke *et al* proposed automated theorem provers for theorems in basic analysis. It is called Analytica and it is written in Mathematica language and run in the Mathematica environment [42]. The process of verifying an expression using Analytica is based on three steps: simplification, inference and rewriting. Given a formula that needs to be proved, first it is passed to the simplification process in order to be reduced by applying a number of algebraic and logical transformation rules. Then, if the simplified result is true, the proof process is terminated. Otherwise the theorem prover matches the formula with the conclusion of some inference rule.

Another project of building a TPS inside CAS is RedLog [32], which is a TPS based on first-order logic built on top of the CAS Reduce. This work extends the Reduce's system by adding a set of symbolic computation methods which provide a lot of rules written in first-order logic. This work is used mainly to simplify Mathematica expression by eliminating the free quantifier of first order logic formulas.

In this approach of building a TPS inside a CAS, TPS is used to guarantee the correctness of certain steps during the simplification process such as the division by a symbolic expression that could be zero. Therefore, in this case, the use of the TPS can prevent common errors done by the symbolic computation system. However, the limitation of this work is that we rely mainly on the CAS and specially Mathematica to check the correctness of the simplification process but not to formally verify a complex hardware system. In addition, similarly to the approach mentioned previously, these approaches are specific to a certain software system and it is a hard task to accommodate or extend them to other systems. Moreover, RedLog relies on a TPS that is based on first-order logic formal system which is undecidable.

1.3.3 Bridge between a CAS and a TPS

This approach is mainly based on the implementation of a protocol of communication between a CAS and a TPS. In many cases, it involves a master-slave relation, in which the TPS is usually considered as a master and the CAS as a slave, with the assumption that there is no trust in the CAS.

In [49], Harrison *et al* propose a bridge between HOL and Maple. This link considers HOL as a master and Maple as a slave. In this work, every result given by Maple will be rigorously certified by HOL. In the cases where we cannot check the correctness of the CAS result, Gordon proposed an intermediate level of trust [16]. It consists of adding a tag to the generated theorem emphasizing that Maple is used. The implementation of this work is decomposed on three components : HOL, Maple and a bridge.

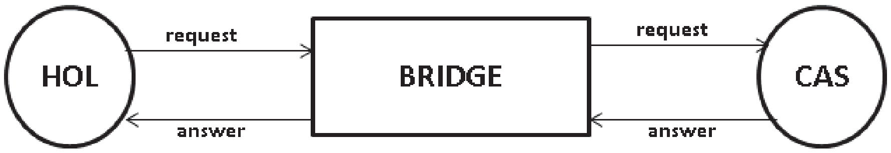


Figure 1.2: Bridge between HOL and Maple [49]

As described in Figure 1.2, HOL and Maple send and receive messages through the Bridge. This Bridge receives and translates the HOL formulas expressed as string into the corresponding Maple encoding and then sends it to Maple. Once the result is ready, it translates and sends it back to HOL. The limitation of this work is the data integration. For example, integrating the CAS AXIOM which has a different structure in Maple is a very difficult procedure because one needs to implement another bridge specific to the syntax of AXIOM.

In [33], Adams *et al* present an interface that connects the Maple (version 6) to PVS. This approach assumes that PVS is used as a slave and Maple as a master. This work allows Maple users to check the correctness of the results during the simplification process by calling PVS as a black box system. It takes advantage of the verification done by PVS without losing the speed and the power of Maple. This approach involves a tightly coupled system where Maple controls PVS. This work is implemented using C programming language and Maple code. First,the user sends a command from the console of Maple to establish the communication between Maple and PVS. Then, the user send the proof command where he specifies the PVS session identify, , the formula that needs verification in PVS syntax, the PVS library and the PVS proof control. Finally, this bridge confirms whether the proof is correct or not.

In [35], Ballarin *et al* propose a bridge connecting Isabelle to Maple. This approach implements an interface between Maple as a slave and Isabelle as a master. This interface consists on extending the Isabelle simplifier by adding new simplification rules in order to call specific operations of Maple. These new simplification rules are written in Isabelle. This implementation does not involve modifying the core of Maple but it should specify its concrete syntax. However, this approach does not allow to access easily Isabelle by another theorem prover or another CAS.

The limitation of these approaches is that they represent a one-to-one connection. Thus, to be able to implement an integration of multiple CASs and TPSs , one needs to adopt another network topology to be able to connect them which may not be an easy task.

1.3.4 Connecting CASs and TPSs using a Mathematical Standard

A mathematical standard is an XML representation of a mathematical expression. The approach of connecting multiple MMSs consists of building a framework that integrates TPSs and CASs. It can be divided into two groups.

The first approach is to build an integrated framework or an MMS (e.g. Mathscheme [52]) that provides the functionalities of both CASs and TPSs without calling any external MMS and without using any intermediate language. This framework is able to play a role as a CAS in order to simplify or evaluate the mathematical expression or as a TPS to proof the correctness of the result. Unfortunately, the implementation of such an MMS requires a lot of effort because it is a creation of two different systems in one. In [41] the authors developed a formal grammar for the MathScheme language where it defines both the algebraic functions and the deductive ones.

The second approach is to build an integrated framework that uses a mathematical standard that can be exchanged between different MMS (such as MathML [14] and OpenMath [22]). Both MathML and OpenMath are standards for representing mathematical expressions, but unlike MathML, OpenMath deals not only with the syntax but also with the semantics of the mathematical object.

In [38] the authors present a JAVA applet which takes a Maple expression as input and returns a Lego [9] theorem prover expression through a translation into OpenMath. The flow of the implementation of this work is represented a follows : first, in the JAVA applet between Lego and OpenMath, the user writes the input statement in Maple syntax. This statement is passed to the Maple Phrasebook which converts it into an OpenMath object. Then, the JAVA applet takes the OpenMath object and returns its corresponding term in Lego syntax using the encodeing/decoding methods existing in the Lego Phrasebook. Finally, Lego proves this statement and sends it back to the JAVA applet.

Our proposed work has been directly inspired by this approach as well as the approach described in Subsection 1.3.3. We propose to combine the access to external tools using an intermediate standard such as OpenMath. In the following section, we describe our proposed methodology.

1.4 Proposed Methodology

We combine the advantages of HOL Light as a TPS and Mathematica as a CAS to create a complete framework that is able to simplify mathematical expressions in a short time and provides a sound result. The architecture of our framework can be generalized in order to connect multiple MMSs together in a way to have access to their kernels. We aim at providing a general architecture to build a heterogeneous problem-solving environment connecting HOL Light, a higher-order logic theorem prover, to any MMS which supports the OpenMath standard.

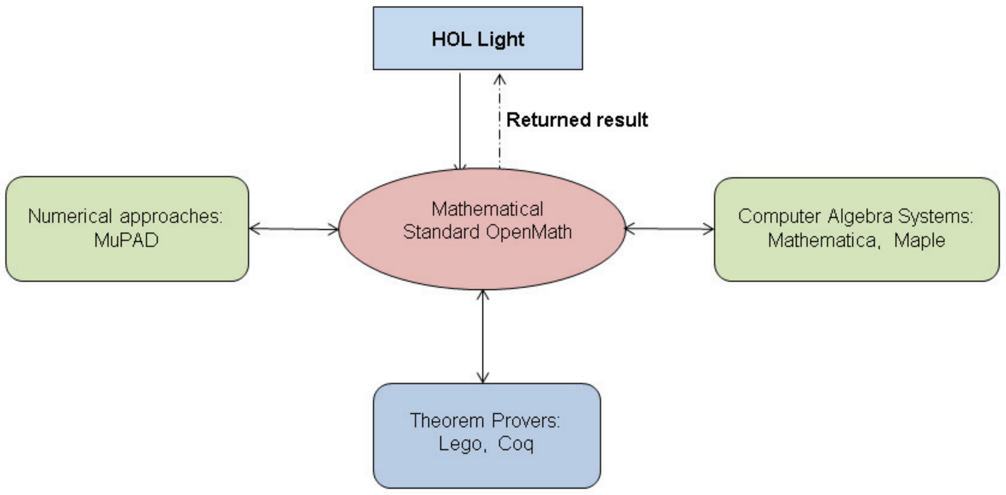

HOL Light
Returned result
Numerical approaches: MuPAD
Mathematical Standard OpenMath
Computer Algebra Systems: Mathematica, Maple
Theorem Provers: Lego, Coq

Figure 1.3: Connecting Different MMS using OpenMath

Figure 1.3 illustrates our approach which provides us with a variety of different MMSs that support Open-Math such as the theorem provers Lego [9] and Coq [25], the CASs Maple, Gap [4] and Mathematica [12], or the numerical solver MuPAD [17] with the intention of solving and reasoning over larger sets of problems. We have implemented a prototype tool of the above approach that links HOL Light to Mathematica as described in Figure 1.4.

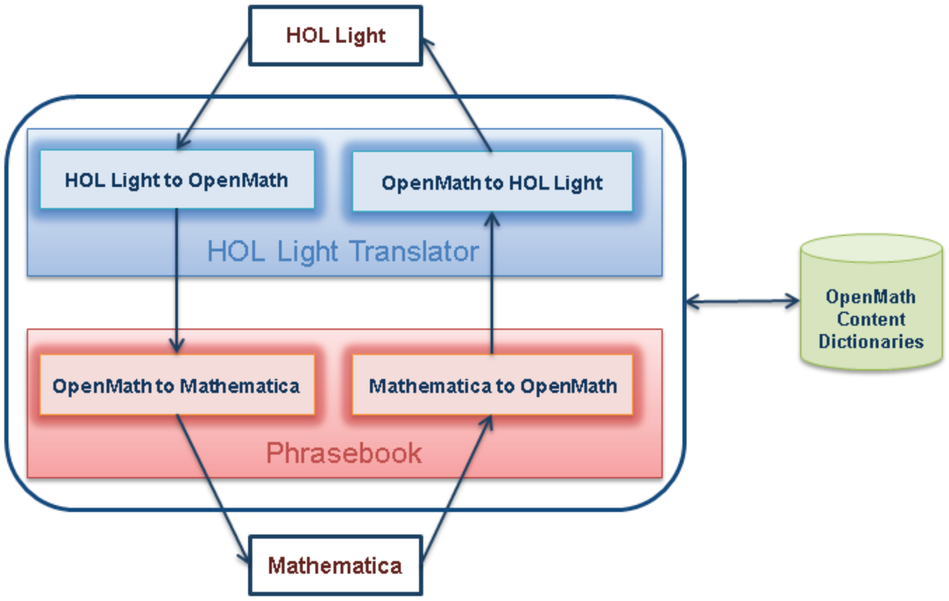


Figure 1.4: Proposed Methodology Linking HOL Light to Mathematica

Figure 1.4 describes the flow of our prototype connecting HOL Light to Mathematica using OpenMath as a middle-ware. Given a HOL Light expression that needs to be simplified, first we pass it to the HOL Light translator and particularly the first module responsible of converting HOL Light expression to OpenMath encoding. Then, we send this OpenMath description to the JAVA application called OpenMath-Mathematica Phrasebook proposed by Caprotti [38]. This Phrasebook is an interface which is compromised of a set of methods converting OpenMath object to/from the internal representation of Mathematica. Once we get the Mathematica input statement, we pass it to the Mathematica kernel in order to be evaluated. After the computation, we translate back the output Mathematica statement to the

OpenMath encoding. This encoding is sent back to the HOL Light translator in order to convert it back to HOL Light syntax. Unlike [38] which implements a JAVA applet where a user should provide a Maple expression in order to be checked by calling Lego through OpenMath, our work offers HOL Light users an easy access to Mathematica. This access involves the connection of these two external tools using the Mathematical standard OpenMath.

1.5 Thesis Contribution

The contributions of this thesis can be summarized as follows:

– We proposed a general approach to combine algebraic computation and deductive reasoning. The use of OpenMath as a middle-ware in our implementation enables a flexible, open and easily extendable architecture. Both CASs and TPSs are connected via an independent mathematical standard language. Instead of the prototype linking HOL Light and Mathematica, we could have chosen any other MMS, different from Mathematica, that supports OpenMath.

– The proposed interface between HOL Light and Mathematica is designed by implementing a HOL Light translator without any modification of the internal implementation of HOL Light. We develop a parser in HOL Light that converts HOL Light syntax to OpenMath encoding. It uses a grammar that can be easily extended to handle more complex mathematical expressions.

– Our methodology combines the soundness of the result proved by TPS, HOL Light and the facility of the simplification of the CAS, Mathematica and builds an integrated environment. In fact, one can call CAS directly in the current running HOL Light session. Then, in a few seconds, the user gets the result simplified by Mathematica and ready to be used for further proofs instead of doing it manually.

– The implemented prototype tool linking HOL Light to Mathematica, called *HolMatica*, is in a perpetual state of improvement as it can be easily extended to tackle more complex mathematical expressions. Our tool is an Open Source software and is publicly available for download from the web site `http://hvg.ece.concordia.ca/research/tools/holmatica/`.

1.6 Thesis Outline

The rest of the thesis is organized as follows. In Chapter 2, we present the Preliminaries. We give an introduction to the theorem prover HOL Light. Then, we discuss the computer algebra Mathematica. Finally, we introduce the notions related to OpenMath and its architecture. Chapter 3 gives a detailed description of the proposed methodology linking HOL Light to Mathematica by showing the functionalities of each modules. We conclude this chapter by presenting the different Mathematica functions that we use in this thesis. Chapter 4 presents the implementation of our tool by describing the important functions of the main parts of our system. Then, we show the usefulness of our tool by presenting several applications and running examples. Chapter 5 provides the conclusion and some future directions to our work.

Chapter 2

Preliminaries

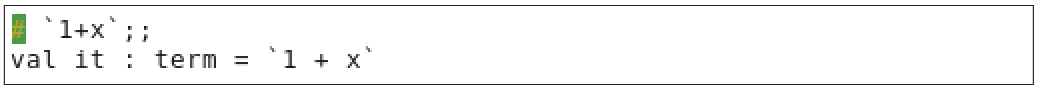
This chapter focuses on the preliminaries needed to set the ground for this thesis. We first give a brief introduction to the theorem prover system HOL Light. Then we describe the computer algebra system Mathematica. Finally, we present the mathematical standard OpenMath and its architecture.

2.1 HOL Light

The HOL Light system is a TPS software that is implemented in Objective Caml (OCaml) [21]. It unifies functional, imperative, and object-oriented programming. OCaml is the main implementation of the Caml programming language. It is derived from the ML programming language family [46]. HOL Light is able to prove mathematical theorems formally expressed in Higher Order Logic (abbreviated as HOL). HOL is a symbolic formal system used in mathematics, philosophy, linguistics, and computer science [57]. It is more expressive than the first-order logic because it has a strong semantic and uses quantifiers over predicate and function symbols. Besides existing basic types in OCaml such as string and integer, HOL Light evaluates expressions involving terms and theorems.

2.1.1 Term

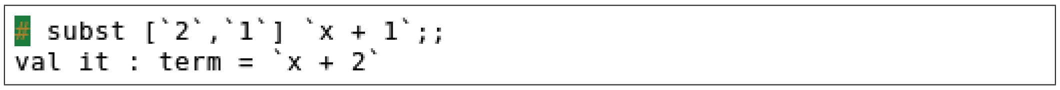
A term is the representation of a mathematical expression or a logical assertion. Unlike strings which are evaluated as symbolic expressions, terms are represented as structures of an abstract syntax tree. Figure 2.1 shows an example of the term $1 + x$ in HOL Light:



```
# `1+x`;;
val it : term = `1 + x`
```

Figure 2.1: Representation of a Term

HOL Light provides a number of operations for manipulating terms. For example the operation `subst` replaces one term by another at all its occurrences. Figure 2.2 shows an example of the function `subst` where we replace the number $'1'$ by the number $'2'$ in the term $'x + 1'$:

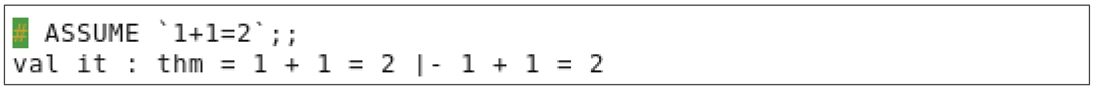


```
# subst [`2`,`1`] `x + 1`;;
val it : term = `x + 2`
```

Figure 2.2: Representation of the Operation of Substitution

2.1.2 Theorem

A theorem represents only true formulas. A formula is a term of Boolean type that may be true or false with respect to given theories. A theorem describes the statements that have been proved by the application of basic axioms and inferences rules on a given formula. The function `ASSUME`, for example, returns a theorem that is deduced from itself. Figure 2.3 shows an example of this function, given a formula $1 + 1 = 2$, it returns the theorem $1 + 1 = 2 \vdash 1 + 1 = 2$.

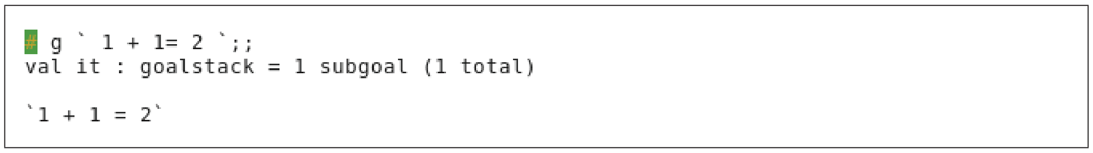


```
# ASSUME `1+1=2`;;
val it : thm = 1 + 1 = 2 |- 1 + 1 = 2
```

Figure 2.3: Representation of a Theorem

2.1.3 Proof Mechanism

In HOL Light the proof derivation can be done in a top-down (forward) mechanism, starting from the axioms, or in a bottom-up (backwards) mechanism, starting from the statement which we want to prove. The second approach is more natural because it decomposes a formula into its sub-formulas. However, it requires the application of inference rules in a reversed way, i.e., from conclusion to the premises. For this reason a special mechanism, called tactics, is integrated in HOL Light. For example, if we want to have as a goal the formula $A(x) \wedge B(x)$, we can call the tactic `CONJ_TAC` that splits our subgoal to two subgoals, namely $A(x)$ and $B(x)$, which will be proved under the same assumptions. The other mechanism, called tactical, allows HOL Light users to compose tactics. For example, `ARITH_TAC` is a tactical which performs a simplification of arithmetical expressions. The tactics and tacticals in HOL Light are implemented as OCaml functions. These tactics are applied on goals which are expressions that the user wants to prove. Figure 2.4 shows an example of a goal which consists of the equality $1 + 1 = 2$:



```
# g ` 1 + 1= 2 `;;
val it : goalstack = 1 subgoal (1 total)

`1 + 1 = 2`
```

Figure 2.4: Representation of a Goal

2.1.4 HOL Light Symbols

Table 2.1 provides the mathematical interpretations of some frequently used HOL Light symbols and functions, which are used in this work.

Table 2.1: HOL Light Symbols

HOL Light Symbol	Meaning
$\wedge$	Logical *and*
$\vee$	Logical *or*
$\sim$	Logical *negation*
&	Cast from Integer to Real
pow	Power Function
λ x. f(x)	Function that maps x to f(x)
real_integral	Integral Function
mat2x2	Matrix of order 2

In this work, we use HOL Light as a TPS because it offers very rich libraries (e.g. real and complex analysis, vector analysis and multivariate calculus). In addition, HOL Light is built on top of OCaml which can be called by external systems, thus it is easier to be combined with other tools, compared to other TPSs such as Isabelle and HOL.

Despite all the advantages of HOL Light, only computer specialists in the theorem proving domain are mostly using it, while mathematicians and scientists often use CAS. This is due to the fact that sometimes formalizing basic theories might require substantial time, knowledge and skills while in these cases, using CAS is simpler. Moreover, sometimes we can encounter mathematical problems that need to solve non-closed form equations. In that case, we need a CAS to simplify or compute the intend equations. Then, we transfer them either manually or automatically to HOL Light to continue further proofs.

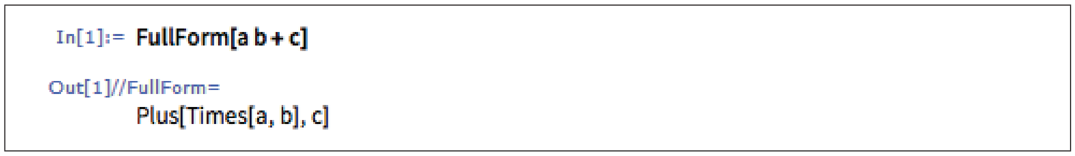
2.2 Mathematica

Mathematica is a product of Wolfram Research, Inc [12]. It is a general computer software system and language that can handle symbolic, numerical and graphical computation. It is under perpetual development (now we find available Mathematica version 9). Compared to other CAS, Mathematica is popular among users [44]. It is used in a large number of solutions in different fields such as Engineering, Finance, Statistics, Business, Biotechnology and Medicine, and other kinds of sciences [2]. In addition, Mathematica can communicate with external programs at a high level and exchange structured data with them using MathLink [6].

MathLink provides a general interface allowing the connection between Mathematica and external tools. The MathLink library consists of a collection of routines that allow external programs either to call Mathematica, and/or to be called by Mathematica by using programming languages such as C/C++ or JAVA. This link can either be on a single computer, or it can be over a network which groups many terminals together. The transparency of the MathLink library makes the link more flexible and independent from any computer. In fact, it emphasizes one of the most important features of MathLink which is interpretability [13].

Mathematica is mainly composed of two parts, the kernel and the front end. The kernel interprets expressions and returns result expressions. The front end provides a GUI, which allows the creation and editing of Notebook documents. Mathematica handles different kinds of mathematical representations such as mathematical formulas, lists and graphics. Although they have a different structure, all these mathematical representations are rendered by Mathematica in one uniform way which is a full form of expression, with no special syntax. This procedure is represented by a function called FullForm.

Figure 2.5 shows an example of a FullForm representation of the mathematical expression $a * b + c$:



```
In[1]:= FullForm[a b + c]

Out[1]//FullForm=
        Plus[Times[a, b], c]
```

Figure 2.5: The FullForm Function

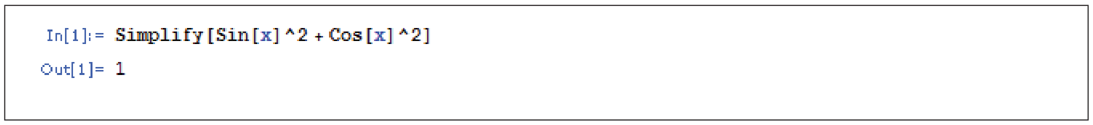
In addition, Mathematica uses algebraic formulas and functions to perform symbolic computation,as well as numbers to perform numerical calculations. In the following, we present the different Mathematica functions that we use in this thesis.

2.2.1 Simplify

Mathematica Syntax:

`Simplify[expression]`

The function `Simplify` performs a sequence of algebraic transformations on the input expression and returns the simplest form it finds. Figure 2.6 shows the result of the simplification of the function $cos^2x + sin^2x$:



```
In[1]:= Simplify[Sin[x]^2 + Cos[x]^2]
Out[1]= 1
```

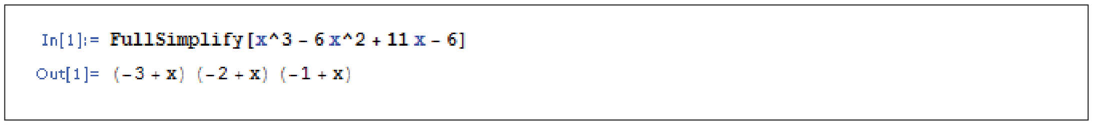
Figure 2.6: The Simplify Function

2.2.2 FullSimplify

Mathematica Syntax:

`FullSimplify[expression]`

The function `FullSimplify` applies elementary and special functions to transform the input expression in order to return the simplest form it finds. Figure 2.7 shows the simplification of the polynomial $x^3 - 6*x^2 + 11*x - 6$, which returns its simplest form $(-3+x)(-2+x)(-1+x)$ after applying the different algebraic transformation:



```
In[1]:= FullSimplify[x^3 - 6 x^2 + 11 x - 6]
Out[1]= (-3 + x) (-2 + x) (-1 + x)
```

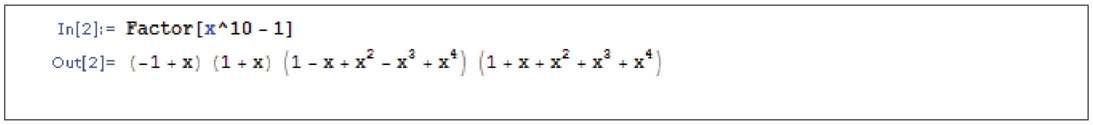
Figure 2.7: The FullSimplify Function

2.2.3 Factor

Mathematica Syntax:

`Factor[polynomial]`

The function `Factor` factors a polynomial over the integers. Figure 2.8 shows the factorization of the polynomial $x^{10} - 1$:



```
In[2]:= Factor[x^10 - 1]
Out[2]= (-1 + x) (1 + x) (1 - x + x^2 - x^3 + x^4) (1 + x + x^2 + x^3 + x^4)
```

Figure 2.8: The Factor Function

2.2.4 FindRoot

Mathematica Syntax:

`FindRoot[f, x, x0]`

The function `FindRoot` searches for a numerical root of a function *f*, starting from the point $x = x_0$. Figure 2.9 finds the approximate values for the function $cosx = x$ for the value close to 0:

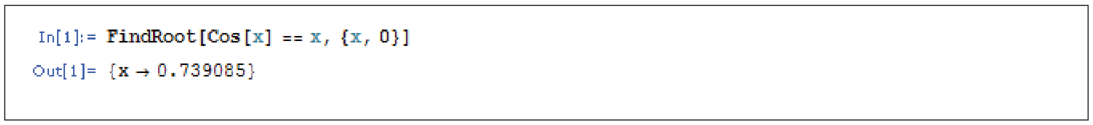

```
In[1]:= FindRoot[Cos[x] == x, {x, 0}]
Out[1]= {x → 0.739085}
```

Figure 2.9: The FindRoot Function

2.2.5 Solve

Mathematica Syntax:

```
Solve[expression, vars]
```

The function `Solve` attempts to solve the expression based on equations or inequalities for the variables vars. Figure 2.10 solve the polynomial $x^2 + ax + 1 = 0$:

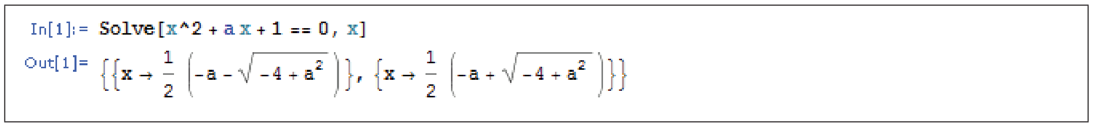

```
In[1]:= Solve[x^2 + a x + 1 == 0, x]
```
Out[1]= $\left\{\left\{x \to \frac{1}{2}\left(-a - \sqrt{-4 + a^2}\right)\right\}, \left\{x \to \frac{1}{2}\left(-a + \sqrt{-4 + a^2}\right)\right\}\right\}$

Figure 2.10: The Solve Function

2.3 OpenMath

2.3.1 OpenMath Definition

OpenMath is a standard that represents mathematical objects with their semantics. It can be used in many applications such as exchanging mathematical information between computer programs. Moreover, it can be stored in databases or published on the worldwide web [22]. In fact, OpenMath can be used not only to express usual mathematical expressions for CASs, but it can also be used to express formulas, theorems and logic expressions that can be exchanged and understood by TPSs.

In the next section, we will present the architecture of OpenMath which consists of OpenMath object, XML encoding, Content Dictionaries and OpenMath Phrasebook.

2.3.2 OpenMath Architecture

Figure 2.11 represents the communication model of how a mathematical expression can be exchanged through OpenMath standard. This architecture is based on three layers. The first layer consists of the internal representation of a specific program. This layer does not concern OpenMath. The second layer is the representation of the first layer as an OpenMath object. In fact, the correspondence between a mathematical object expressed in its internal representation and its OpenMath object is performed via OpenMath Phrasebook [40]. The OpenMath Phrasebook is an interface that converts an OpenMath object to/from the internal representation in a software application. This conversion is governed by the corresponding content dictionaries (CDs) [18]. These CDs contain the definitions of symbols occurring in the OpenMath objects. The third layer is a representation of the corresponding OpenMath object as a byte stream that can be used to communicate with external systems [54].

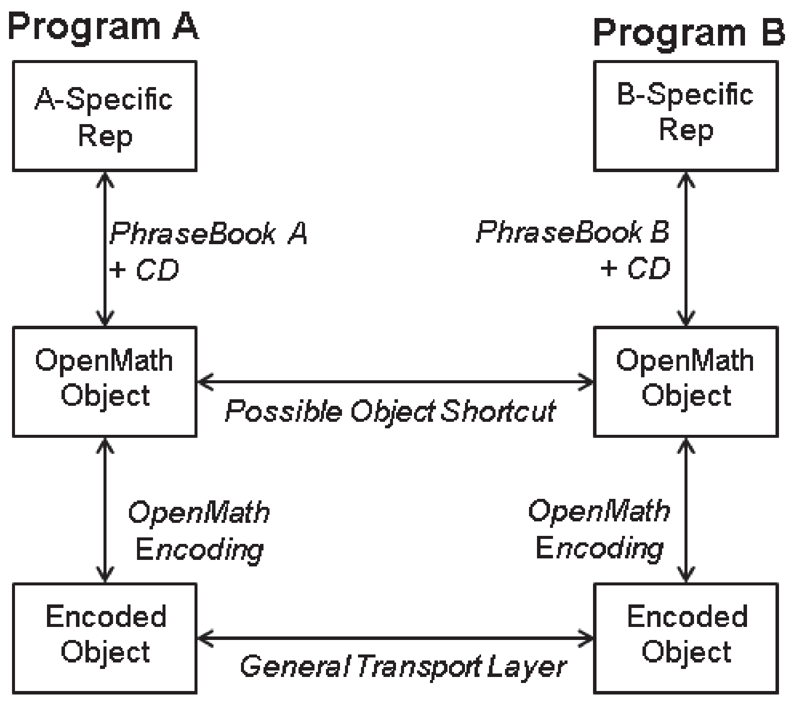


Figure 2.11: Architecture of OpenMath [43]

2.3.2.1 OpenMath Object

OpenMath object is the representation of both the syntax and the semantics of a mathematical expression that can be exchanged between several software systems [43]. OpenMath provides two categories of OpenMath objects. The first is the basic object that describes integers, symbols, floating-point numbers, character strings, byte arrays and variables. The second one is the compound objects [53]. Compound objects are described using:

– Application objects: can represent either a functional application such as `application(cos, x)` or a constructor `application(Rational, 1,2)`.

– Binding objects are represented as `Binding (lambda, x , application(times, 1, x))`.

– Error objects can occur during the manipulation of the OpenMath such as `error(damaged encoding)`.

– Attribution objects are represented as `attribution (A, type t)` which expresses that the object A has a type t.

2.3.2.2 OpenMath XML Encoding

The XML encoding of an OpenMath object describes its representation in stream bytes that can be exchanged easily between external systems or stored and extracted from files [54]. For instance, the encoding of:
`Binding(lambda, x , application(times, 1, x))` is described as follows:

```
<OMOBJ><OMBIND><OMS name="lambda" cd="fns"/>
            <OMBVAR><OMV name= "x"/></OMBVAR>
            <OMA> <OMS name="times" cd="arith1"/>
                 <OMI> 1 </OMI>
                 <OMV name= "x"/>
            </OMA>
</OMBIND></OMOBJ>
```

This encoding describes that the symbols `lambda` and `times` (tagged by `OMS`) are defined by the CDs `fns` and `arith1`, respectively. In addition, the elements `OMA`, `OMBIND`, `OMI` and `OMV` identify `application`, `binding`, `integer`, and `variable`, respectively. [39]

2.3.2.3 Content Dictionaries

A Content Dictionary (CD) represents the semantics of a mathematical expression independently of the application. It contains several details about the CD itself such as name, status (official, experimental, private or obsolete) and an optional list

of the CDs that it may depend on. A CD describes also a collection of symbols, their designations, their descriptions in natural language and rules which define the use of appropriate symbols in the correct order. In addition, we can add optional information related to the specific symbol such as the signature, the proprieties and the example explaining the use of this specific symbol [48].
The following example represents the CD of the transcendental `"sin"` function :

```
<CDDefinition>
 <Name> sin </Name>
 <Description>
 The sin function as described in Abramowitz and Stegun, section 4.3
 </Description>
</CDDefinition>
```

This definition allows the mapping implementation of the OpenMath object `<OMS name="sin" cd="transc"/>` to its internal representation via Phrasebook in the correct order.

2.3.2.4 OpenMath Phrasebooks

Phrasebook is an interface which converts OpenMath object to/from the internal representation of a program as decoded by the related CDs [38]. Since the emergence of OpenMath, several Phrasebooks have been implemented. Mostly those Phrasebooks are used to exchange OpenMath object using CASs such as GAP, Axiom, Mathematica, Maple, etc. Moreover, we can find other Phrasebooks for exchanging OpenMath objects using proof checkers such as Lego and Coq or numerical approach such as MuPAD. In our work we use OpenMath-Mathematica Phrasebook proposed by Caprotti *et al.* [40].

2.3.3 Summary

In this chapter we provided an introduction to HOL Light theorem prover. We presented an overview of Mathematica. Then, we described the OpenMath standard and its architecture. The intent was to introduce basic notions that are going to be used in the rest of the thesis. In the next chapter, we present our methodology linking HOL Light to Mathematica through OpenMath.

Chapter 3

Linkage Methodology

In this chapter, we present a detailed description of the proposed methodology linking HOL Light as a TPS to Mathematica as a CAS. Then, we describe the functionalities of each module. Finally, we conclude this chapter by giving some examples of the different Mathematica operations supported in our work.

3.1 Methodology Overview

Figure 3.1 depicts our methodology to connect HOL Light and Mathematica. This is comprised of two modules: the OCaml units and the JAVA application. It includes also the XML objects which represent the OpenMath objects that are exchanged between the two modules. It can be observed in Figure 3.1 that the connection between each pair of modules is bidirectional. This makes the connection between HOL Light and Mathematica complete.

The steps of our methodology are described as follows. First, we pass the HOL Light expression that needs to be simplified to the *"Parser & Splitter"*.

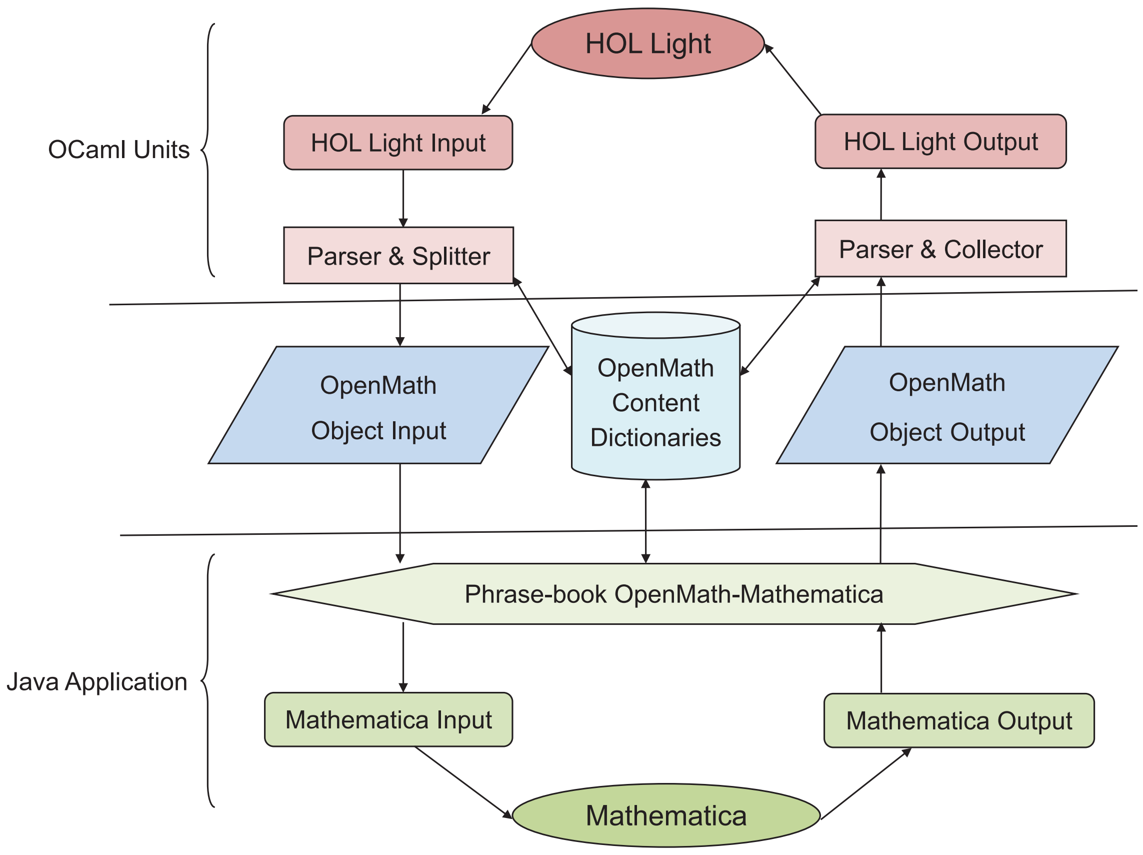


Figure 3.1: Flow of Connecting HOL Light to Mathematica

At this stage, we use the *"Parser & Splitter"* unit in order to transform the HOL Light term into a corresponding OpenMath object. This conversion is governed by the relevant CDs. Then, we store the OpenMath XML encoding which represents the input of the second module that contains the JAVA application. This JAVA application contains the OpenMath-Mathematica Phrasebook [40], which is an interface responsible for three tasks. First, it reads the OpenMath XML encoding and translates it into Mathematica syntax as understood by means to the corresponding CDs. Second, it passes the Mathematica statement to Mathematica kernel using MathLink in order to evaluate the statement. Finally, after the computation of the intended equation, the Mathematica output statement is translated back to OpenMath and an XML representation is generated.

The same process takes place in the reverse direction until we reach the *"Parser & Collector"* module which translates back the OpenMath object into the HOL Light statement according to the corresponding CDs.

Our methodology takes the HOL Light input expression as one string and the Mathematica expression as another string. By default the returned value is a HOL Light theorem tagged with the name of the CAS, in our case Mathematica. As an example, let's assume that the returned result is Ψ. The tag is then represented as : `Mathematica` $\vdash \Psi$. This tag intends that *"what we get from Mathematica is sound"*. Moreover, each theorem derived from this returned theorem inherent the tag `Mathematica`. This procedure helps HOL Light users to easily trace the theorems created by the help of the external tool. In addition, we can generate a sub-goal as a returned result. In this way, after the computation, the returned result is represented as a sub-goal and added to the assumptions of the main goal. One needs to prove this sub-goal in order to pursue further proofs. In this case the soundness of our result is preserved as we are using HOL Light to prove it.

In the next section, we describe the functionalities of the different modules presented in the methodology.

3.2 OCaml Unit

This module is responsible of the translation from HOL Light to OpenMath and vice-versa. It consists of two OCaml units : *"Parser & Splitter"* which translates HOL Light to OpenMath XML encoding and the *"Parser & Collector"* which translates OpenMath objects into HOL Light statement. It contains also the HOL Light input and output statements.

3.2.1 HOL Light Input Statement

The HOL Light input statement has two arguments, i.e., the HOL Light expression needed to be simplified as a string and the specific Mathematica function e.g., `Simplify` as a string.

3.2.2 HOL Light Output Statement

The HOL Light output Statement can be either a theorem or a sub-goal. Once the returned result described in XML OpenMath encoding is translated back in HOL Light, by default a theorem tagged with the name of Mathematica is generated. In case we need to integrate the result in a main goal, it generates a sub-goal.

3.2.3 Parser & Splitter

The *"Parser & Splitter"* module first parses and decomposes the HOL Light input statement into a list of operators and operands using the function *lexical_analyzer*, as described below. This function takes the HOL Light input statement and returns the list of the used symbols. Then, the procedure *map_symbols* maps each element of the list with the corresponding OpenMath symbol as decoded by means of the related CDs. Finally, using the function *write_object*, the *"Parser & Splitter"* stores the OpenMath XML encoding. This representation can be exchanged among several systems. The XML description is used as an input for the second module, as we mentioned previously. In addition, the *"Parser & Splitter"* unit saves the tag of the specific Mathematica function in order to be passed to the JAVA application.

Parser & Splitter Functionality

```
Input:  hol_light_expr= HOL Light expression as a string
        // Represents the equation to be computed
Output: OpenMath_Input_object = OpenMath XML encoding object
begin
    $list\_of\_tockens \longleftarrow lexical\_analyzer(hol\_light\_expr)$;
    // Decomposes the input into a list of lexical items
    $OpenMath\_objects \longleftarrow map\_symbols(list\_of\_tockens)$;
    // Maps HOL Light symbols with OpenMath objects
    $OpenMath\_Input\_object \longleftarrow write\_object(OpenMath\_objects)$;
```

3.2.4 Parser & Collector

The *"Parser & Collector"* module, first reads the OpenMath XML encoding object of the returned result from Mathematica and extracts the OpenMath symbols, using the function *extract_OM_Obj*, as described below. This function takes the OpenMath XML encoding and returns the list of the OpenMath used symbols. Then, the *"Parser & Collector"* translates these OpenMath symbols into a HOL Light statement as understood by the relevant CDs. This step is performed using the function *map_HL_symbol* which maps the OpenMath symbols to the specific corresponding HOL Light symbols. Then, it collects all the HOL Light symbols using the function *collect_HL_symbols*. Finally, it returns the HOL Light output statement.

Parser & Collector Functionality

```
Input: OpenMath_Output_object = OpenMath XML encoding object of
       Mathematica output expression

Output: hol_light_res= HOL Light expression as a string
begin
    $list\_OM\_obj \longleftarrow extract\_OM\_Obj(OpenMath\_Output\_object)$;
    // Reads the OpenMath XML encoding input object
    $List\_HLL\_symbols \longleftarrow map\_HL\_symbols(list\_OM\_obj)$;
    // Maps OpenMath objects with HOL Light symbols
    $hol\_light\_res \longleftarrow collect\_HL\_symbols(List\_HL\_symbols)$;
    // Collects the HOL Light Symbols
    return hol_light_res
```

3.3 JAVA Application Unit

The JAVA application unit is composed of the Mathematica input and output statements and the OpenMath-Mathematica Phrasebook.

3.3.1 Mathematica Input/Output Statements

The Mathematica input/output statements represent the statement in Mathematica language in a full form description which are sent back and forth from the OpenMath-Mathematica Phrasebook to the Mathematica kernel.

3.3.2 OpenMath-Mathematica Phrasebook

In this work, we use the concept of the Phrasebook between OpenMath and Mathematica which was introduced by Caprotti *et al.* [40].

This Phrasebook defines a collection of JAVA classes. It provides a collection of encoding and decoding methods between OpenMath and Mathematica based on the declaration of the corresponding CDs. This Phrasebook includes also a set of methods that handle the Mathematica calling function that the user specifies in the HOL Light input statement.

```
Mathematica-OpenMath Phrasebook Functionality
Input: OpenMath_input_object = OpenMath XML encoding object of
       HOL Light input expression
Output: OpenMath_Output_object = OpenMath XML encoding object
        of Mathematica output expression
begin
    $Mathematica\_input \longleftarrow$
    $translate\_OM\_Mathematica(OpenMath\_Output\_object)$;
    // Translates Mathematica statement into OpenMath objects
    $Mathematica\_output \longleftarrow call\_Mathe\_kernel(Mathematica\_input)$;
    // Evaluates the equation by calling Mathematica kernel
    $OM\_result \longleftarrow translate\_Mathe\_OM(List\_HL\_symbols)$;
    // Translates back OpenMath object to Mathematica
    $OpenMath\_Output\_object \longleftarrow write\_object(OM\_result)$;
    // Writes the output OpemMath XML encoding object
```

We first translate the XML encoding that describes the OpenMath input object into a Mathematica statement, as described above. This step is performed using the function *translate_OM_Mathematica* which takes as input the OpenMath XML encoding and returns the corresponding Mathematica statement. Then, we evaluate it through a connection with the MathLink link which opens the Mathematica kernel. In fact, the function *call_Mathe_kernel*, which takes as input the Mathematica input statement and returns its simplification, establishes a connection between this JAVA application with the Mathematica kernel in order to use Mathematica as a computation engine. Once the computation is complete, we translate back the Mathematica Output to OpenMath object, using the function *translate_Mathe_OM*. Finally, we store its XML Encoding which will be read by the module OCaml Units *"Parser & Collector"*, using the last function called *write_object*.

In the following section, we present the main program of our work with a detailed explanation of the different Mathematica functions that we are using in our work.

3.4 Mathematica Functions

In this section, we present the Mathematica functions supported in our framework. These functions, which are specified by the user in the HOL Light input statement contain the same structure, but the way of displaying the returned result is different. In the following subsections, we present details of the Mathematica functions we use in our work, illustrated by some examples.

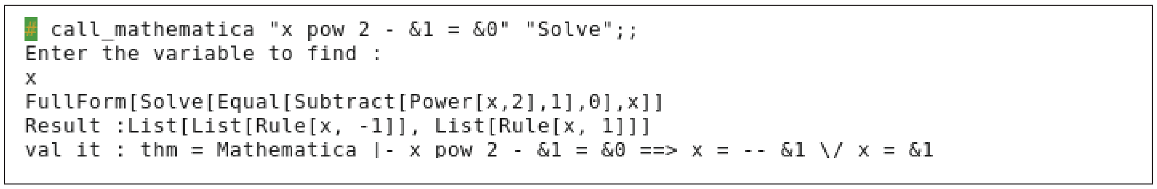
3.4.1 Solving Equations

Solving equations is the process of finding values that respect a specific algebraic system. Given the equation $x^2 - 1 = 0$, we solve it in our work as follows:

```
Input:
#call_mathematica "x pow 2 - &1 = &0" "Solve";;
Output:
val it : thm = Mathematica ⊢ x pow 2 - &1 = &0 ==> x = − &1 \/ x = &1
```

Figure 3.2 describes the different steps to solve the above equation. First, in the HOL Light session, we call the main function of our tool, *call_mathematica*, given the arguments the HOL Light expression `x pow 2 - &1 = &0` where `&1` and `&0` are

type casted `1` and `0` to reals , `x` is a variable, `2` is an integer and `pow` is the power function, and the specific Mathematica function `Solve`. Second, we specify the variable that we want to solve, in our case it is `x`. Then, this statement is translated to a Mathematica statement, expressed as
`FullForm[Solve[Equal[Subtract[Power[x,2],1],0],x]]`, which represents the full form of $x^2 - 1 = 0$. Once the computation is finished, the returned Mathematica statement is represented as
`List[List[Rule[x,-1],List[Rule[x,1]]]]` which describes the set of the two solution as follows : {{ **x -> 1** },{ **x-> -1**}}. Finally, the returned theorem exposes the disjunction of the possible values that `x` can take.



```
# call_mathematica "x pow 2 - &1 = &0" "Solve";;
Enter the variable to find :
x
FullForm[Solve[Equal[Subtract[Power[x,2],1],0],x]]
Result :List[List[Rule[x, -1]], List[Rule[x, 1]]]
val it : thm = Mathematica |- x pow 2 - &1 = &0 ==> x = -- &1 \/ x = &1
```

Figure 3.2: Execution Window for Solving an Equation

3.4.2 Factoring polynomials

Factoring a polynomial represents the product in its irreducible factors. An example where we can factorize the expression $x^3 + 2x^2 + x$ is as follows :

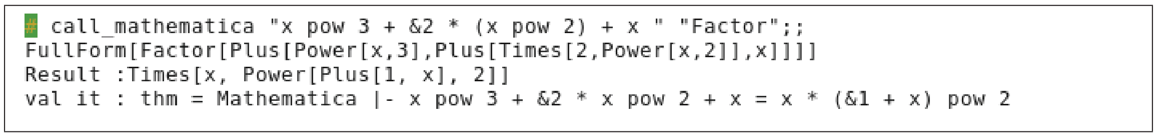
`Input:`

#call_mathematica "x pow 3 + &2 * (x pow 2) + x" "Factor";;

`Output:`

val it : thm = Mathematica $\vdash$ x pow 3 + &2 * x pow 2 + x = x * (&1 + x) pow 2

Figure 3.3 describes the different steps to conduct factoring a polynomial.



```
# call_mathematica "x pow 3 + &2 * (x pow 2) + x " "Factor";;
FullForm[Factor[Plus[Power[x,3],Plus[Times[2,Power[x,2]],x]]]]
Result :Times[x, Power[Plus[1, x], 2]]
val it : thm = Mathematica |- x pow 3 + &2 * x pow 2 + x = x * (&1 + x) pow 2
```

Figure 3.3: Execution Window for Factorizing an Equation

3.4.3 Find Root Equations

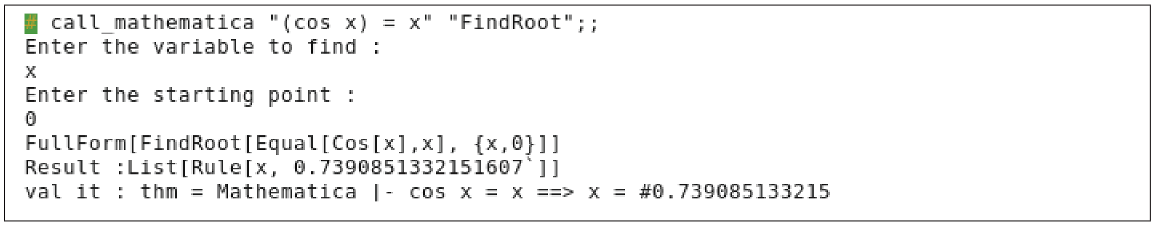
The numerical method `FindRoot` represents finding a value x such that $f(x) = 0$ where f is the given function and x is the root of the function f. For example, we can find a root of the following equation (if it exists) $cos\ \ x = x$:

`Input:`

#call_mathematica “(cos x) = x” “FindRoot”;;

`Output:`

val it : thm = Mathematica ⊢ cos x = x ==> x = #0.739085133215



```
# call_mathematica "(cos x) = x" "FindRoot";;
Enter the variable to find :
x
Enter the starting point :
0
FullForm[FindRoot[Equal[Cos[x],x], {x,0}]]
Result :List[Rule[x, 0.7390851332151607`]]
val it : thm = Mathematica |- cos x = x ==> x = #0.739085133215
```

Figure 3.4: Execution Window for Finding the Root of an Equation

As it can be observed in Figure 3.4 this computation means that we want to search for a numerical root of the equation $cos\ \ x = x$, starting from the point $x = 0$.

3.4.4 Simplifying Equations

The simplification of mathematical equations consists of applying a sequence of algebraic transformations to obtain a simpler result. The different steps to simplify an equation sent by HOL Light to Mathematica are described in the sequel.

The simplification program takes as input the HOL Light expression and returns a HOL Light theorem (or a sub-goal). It starts by calling the *"Parser & Splitter"* module to translate the HOL Light input into OpenMath object. Then, it calls the OpenMath-Mathematica Phrasebook from the evaluation. It returns the result as a string by calling the *"Parser & Collector"* module. Finally, it generates a theorem based on the returned result, i.e., if the result is true, it returns the HOL Light input statement as a theorem (axiom), otherwise it returns its negation. In case the HOL Light input statement is not a relational equation, we create the implication of the input and output and return it as a theorem.

```
Simplification Program
Input: hol_light_expr= HOL Light expression as a string
      // Represents the equation to be computed
Output: HOL Light theorem
begin
    $OpenMath\_Input\_object \longleftarrow call\_Parser\_Splitter(hol\_light\_expr)$;
    // Generates the OpenMath XML encoding of hol_light_expr
    $OpenMath\_Output\_object \longleftarrow$
    $call\_Phrasebook(OpenMath\_Input\_object)$;
    // Evaluation by Mathematica
    $hol\_light\_res \longleftarrow call\_Parser\_Collector(OpenMath\_Output\_object)$;
    // HOL Light expression of the result
    if $hol\_light\_expr$ is a relation then
        if $hol\_light\_res = true$ then
            return new_axiom(hol_light_expr);
            // Returns the expression as a theorem
        else
            return new_axiom($\neg$ (hol_light_epxr));
            // Returns its negation as a theorem
    else
        $HOLLight\_final\_result \longleftarrow hol\_light\_expr <=> hol\_light\_res$ ;
        return new_axiom(HOLLight_final_result);
        // Returns the final evaluation
```

So far, we are able to implement different functions such as simplifying integrals, transcendental functions, computing eigenvectors, finding roots and factorization of complex polynomials. The Mathematica functions for the simplification are `Simplify` or `FullSimplify`. They apply a sequence of mathematical rules that reduce the complexity of a given equation. Unlike `Simplify`, `FullSimplify` tackles a wide set of complex mathematical expressions and returns the simplest result [3]. For example, we can simplify the following expression $x^2 + x + 1 > x^2 + x + 2$:

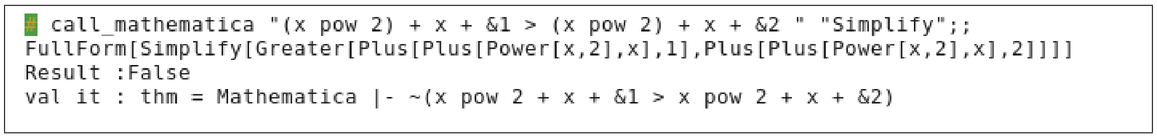
`Input:`

#call_mathematica "(x pow 2) + x + &1 > (x pow 2) + x + &2 " "Simplify";;

`Output:`

val it : thm = Mathematica $\vdash$ ~(x pow 2 + x + &1 > x pow 2 + x + &2)

Figure 3.5 describes the different steps for evaluating a relational equation.



```
# call_mathematica "(x pow 2) + x + &1 > (x pow 2) + x + &2 " "Simplify";;
FullForm[Simplify[Greater[Plus[Plus[Power[x,2],x],1],Plus[Plus[Power[x,2],x],2]]]]
Result :False
val it : thm = Mathematica |- ~(x pow 2 + x + &1 > x pow 2 + x + &2)
```

Figure 3.5: Execution Window for Evaluating a Relational Equation

We can also compute the bounded integral of the mathematical expression $x + 1$ using `FullSimplify` as follows $\int_0^1 (x + 1)\, \mathrm{d}x$.:

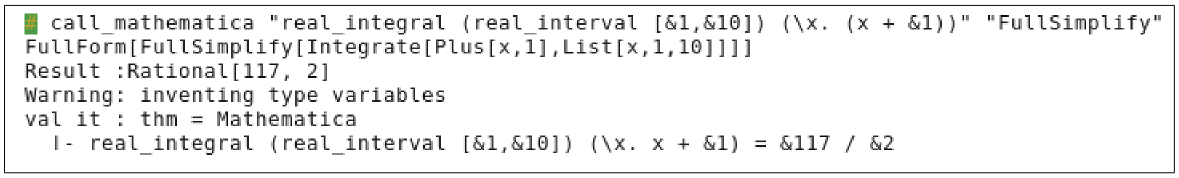
`Input:`

#call_mathematica " real_integral (real_interval [&1,&10]) (\ x. x + &1) " "FullSimplify";;

`Output:`

val it : thm = Mathematica $\vdash$ real_integral (real_interval [&1,&10]) (\ x. x + &1) = &117 / &2

Figure 3.6 describes the different steps for simplifying above bounded integral equation.



```
# call_mathematica "real_integral (real_interval [&1,&10]) (\x. (x + &1))" "FullSimplify"
FullForm[FullSimplify[Integrate[Plus[x,1],List[x,1,10]]]]
Result :Rational[117, 2]
Warning: inventing type variables
val it : thm = Mathematica
  |- real_integral (real_interval [&1,&10]) (\x. x + &1) = &117 / &2
```

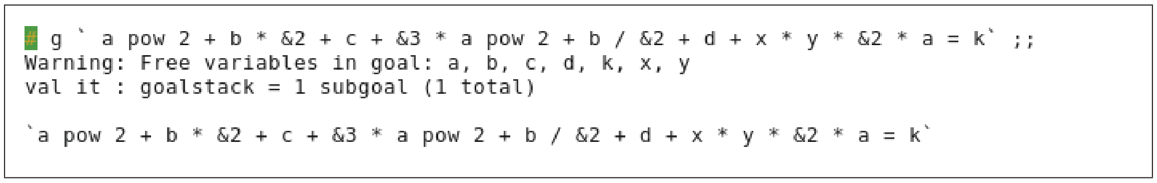
Figure 3.6: Execution Window for Evaluating a Bounded Integral Equation

An example where we define a goal with a complex arithmetics in HOL Light is given below.

`Goal:`

g ‘ a pow 2 + b * &2 + c + &3 * a pow 2 + b / &2 + d + x * y * &2 * a = k‘ ;;

This goal represents a mathematical expression with a large number of arithmetic operations $a^2 + b * 2 + c + 3 * a^2 + b/2 + d + x * y * 2 * a = k$. Figure 3.7 shows the definition of a goal in HOL Light.



```
# g ` a pow 2 + b * &2 + c + &3 * a pow 2 + b / &2 + d + x * y * &2 * a = k` ;;
Warning: Free variables in goal: a, b, c, d, k, x, y
val it : goalstack = 1 subgoal (1 total)

`a pow 2 + b * &2 + c + &3 * a pow 2 + b / &2 + d + x * y * &2 * a = k`
```

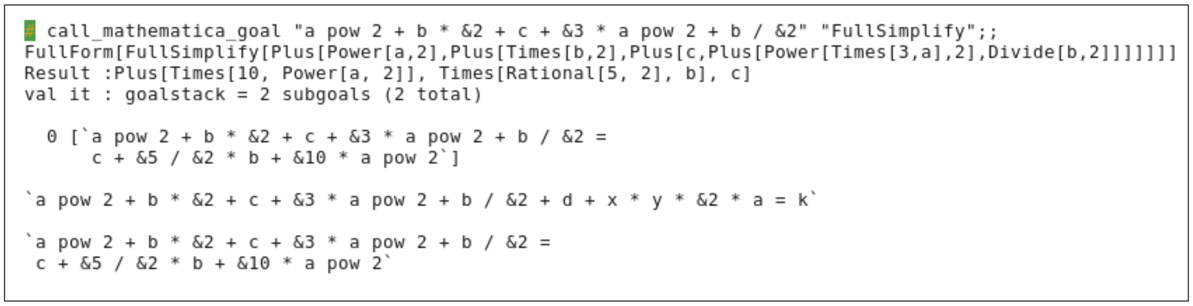
Figure 3.7: Definition of a Goal in HOL Light

Solving this equation in HOL Light may take a substantial time and huge effort. Therefore, we propose to evaluate the sub-term $(a^2 + b * 2 + c + 3 * a^2 + b/2)$ in Mathematica and get the returned result as a sub-goal. We evaluate this expression as follows:

`Input:`

call_mathematica_goal “a pow 2 + b * &2 + c + &3 * a pow 2 + b / &2” “FullSimplify”;;

Figure 3.8 shows that the result $a^2+b*2+c+3*a^2+b/2 = c+5/2*b+10*a^2$ is added in the assumptions of the main goal. Thus, the HOL Light user needs to prove this returned sub-goal in order to prove the main goal. In this case, we do not trust the results from Mathematica. In fact, Mathematica is used only as a computation engine and we rely on HOL Light to prove the correctness of the result.



```
# call_mathematica_goal "a pow 2 + b * &2 + c + &3 * a pow 2 + b / &2" "FullSimplify";;
FullForm[FullSimplify[Plus[Power[a,2],Plus[Times[b,2],Plus[c,Plus[Power[Times[3,a],2],Divide[b,2]]]]]]]
Result :Plus[Times[10, Power[a, 2]], Times[Rational[5, 2], b], c]
val it : goalstack = 2 subgoals (2 total)

  0 [`a pow 2 + b * &2 + c + &3 * a pow 2 + b / &2 =
      c + &5 / &2 * b + &10 * a pow 2`]

`a pow 2 + b * &2 + c + &3 * a pow 2 + b / &2 + d + x * y * &2 * a = k`

`a pow 2 + b * &2 + c + &3 * a pow 2 + b / &2 =
 c + &5 / &2 * b + &10 * a pow 2`
```

Figure 3.8: Execution Window for Simplifying a goal from Mathematica

3.5 Summary

In this chapter, we presented our methodology by explaining in detail its modules. Then, we gave an overview about the Mathematica function supported in our framework. In the following chapter, we will present the technical aspects of our tool by describing our implementation.

Chapter 4

Tool Implementation

In this chapter, we first present an overview of the implementation of our tool. Second, we describe the common data types that we defined. Then, we give details about the implementation of the *"Parser & Splitter"*. We describe also the connection between HOL Light translator and OpenMath-Mathematica Phrasebook. Then, we present the *"Parser & Collector"* implementation. Finally, we conclude this chapter by presenting some running examples of our tool.

4.1 Implementation Overview

Figure 4.1 depicts our process implementing the link between HOL Light to Mathematica. This is comprised of three modules: *"Parser & Splitter"*, OpenMath-Mathematica Phrasebook and *"Parser & Collector"*.

The flow of our implementation is described as follows: First, it translates the HOL Light statement into OpenMath object based on the parsing and mapping functions, described in Section 4.1.2. Second, the JAVA Phrasebook reads the XML file which contains the OpenMath object and converts it to Mathematica expression. The latter is passed to Mathematica with specifying the calling Mathematica function. The computation of Mathematica is translated back to OpenMath object using again the JAVA Phrasebook, presented in Chapter 3. Finally, the returned result from Mathematica is parsed by our tool and converted to a HOL Light theorem.

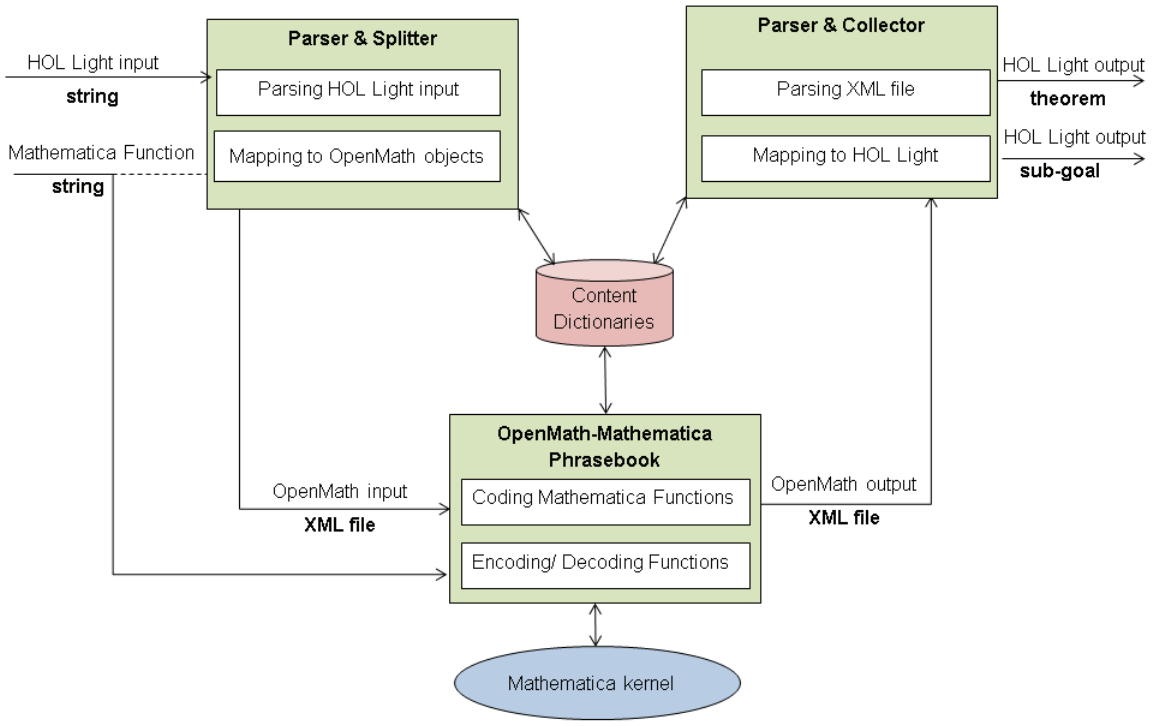


Figure 4.1: Methodology

In the following sections, we describe details of the tool implementation.

4.1.1 Common Data Type Structure

The HOL Light translator converts HOL Light expressions into OpenMath XML encoding and vice-versa. In the implementation of both the *"Parser & Splitter"* and the *"Parser & Collector"*, we used the basic and the compound OCaml data types. Examples of basic types are string, float and int (represents integer). For the compound type, we defined two types: type lexeme and type expression as described in the following subsection.

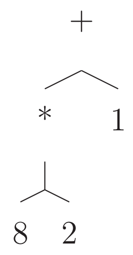
4.1.1.1 Type Lexeme

The type lexeme includes "real" tokens such as numbers with type int or float, variables with the type string and function and predicate symbols with the type string which represent mathematical operations. Also it includes a list of tokens and a special symbol Nil as an end of a list (used in the recognition of a bracket). As an example, the expression `List[ Num 1; Symb '+'; Num 2; Nil]` corresponds to the mathematical expression `(1+2)`.

```
type lexeme =
  | Nil
  | Numf of float
  | Num of int
  | Str of string
  | Symb of string
  | List of lexeme list
```

4.1.1.2 Type Expression

Type expression is a recursive type representing a mathematical expression in a form of a tree. As an example, we consider the mathematical expression : `1 + 2 * 8`. Its tree representation is :



```
type expression =
  | Nbf of float
  | Nb of int
  | Stri of string
  | Eigenvect of expression
  | Eigenval of expression
  | Gt of expression * expression
  | Egt of expression * expression
  | Ls of expression * expression
  | Els of expression * expression
  | Eq of expression * expression
  | Neq of expression * expression
  | Add of expression * expression
  | Sub of expression * expression
  | Mul of expression * expression
  | Div of expression * expression
  | Pow of expression * expression
  | Sin of expression
  | Cos of expression
  | Tan of expression
  | Abs of expression
  | Sqrt of expression
  | Det of expression
  | Integrate_finite of expression * expression
  | Intervale of expression * expression
  | Lambda of expression * expression
```

In the next subsection, we present the implementation of the OCaml *"Parser & Splitter"*.

4.1.2 Parser & Splitter

The *"Parser & Splitter"* transforms the HOL Light statement into a corresponding OpenMath object as understood by means of the CD.

```
let  res =
openmath_expr (syntax (unaries (purge (split (lex str))))) in
  let r= "<OMOBJ>"^res^"</OMOBJ>" in write_file r
```

The first step of this *"Parser & Splitter"* is to parse the string HOL Light expression according to the grammar presented in Figure 4.2. Second, we perform the lexical analysis and generate the lexical items (tokens). This grammar handles many arithmetical terms in HOL Light expressed as statements in higher-order logic, however it is not complete. The *"Parser & Splitter"* decomposes the HOL Light input statement into lexical items. For example, the list of lexical items for the HOL Light expression `5 + 7` which contains the integers `5` and `7` and the binary function `+` is `lexeme list = [ Num 5; Symb '+'; Num 7; Nil]`. The type of these lexemes is already explained above (cf. Section 4.1.1.2). After that, the *"Parser & Splitter"* maps each lexeme of the list to the OpenMath corresponding symbol as understood by means of the related CD. Finally, it stores the description of the OpenMath object in an XML file.

Table 4.1, which describes details of the internally used functions in the *"Parser & Splitter"* module.

Table 4.1: Parser & Splitter Internal Functions

Fonction Name	Type	Tasks
lex	string -> lexeme list	Lexical analyzer: - Takes as input a string - Gives a list of lexical items
split	lexeme list -> lexeme list	- Splits the lexeme list according to the parentheses - Replaces any pair of brackets by lexeme list
purge	lexeme list -> lexeme list	- Removes all Nil lists of lexical items (including those in sub-lists)
Prior_operation	lexeme list -> lexeme list	- Defines the priority between the unary, binary and the prefix operation
syntax	lexeme list -> expr	- Converts the list of tokens to the corresponding expression
openmath_expr	expr -> string	Printer: -Takes as input an expression of operations - Gives its description in OpenMath XML Encoding.

```
hol_atom ::= float
    |integer
    |variable
    |(hol_expr)
hol_expr :: = hol_atom
    |unary_func (hol_expr)
    |hol_expr binary_func hol_expr
    |hol_expr binary_relation hol_expr
    |prefix_operation

prefix_operation ::=Matrix2(hol_expr)
    |Integrate_finite (hol_expr, hol_expr )
    |real_interval(hol_expr ,hol_expr )
    |Eigenval(hol_expr)
    |Eigenvect(hol_expr)
    |Determinant(hol_expr)
    |Inverse(hol_expr)
    |Lambda (variable, hol_expr )
    |vector(list_of_hol_expr)

unary_func ::= sqrt | exp | abs | sin | cos
          |tan  | arcsin | arccos | arctan
binary_func ::= + | - | / | * |  pow
               ++| -- | // | ** | %
binary_relation ::= < | = | > | => | =< | !=
```

Figure 4.2: Lexical Grammar

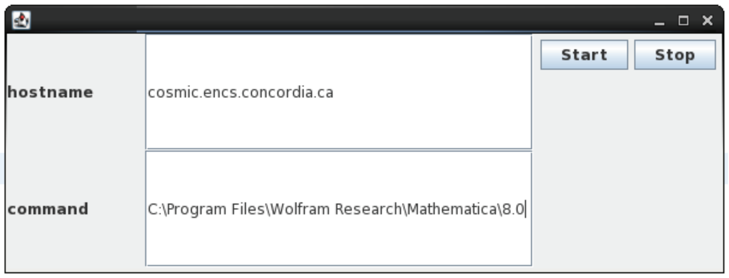
4.1.3 OpenMath-Mathematica Phrasebook

The OpenMath-Mathematica Phrasebook (available from the MathdoxWeb site [11]) is responsible for three tasks. First, it translates the OpenMath input XML Encoding into Mathematica statement. Second, it calls the Mathematica kernel via MathLink and passes the Mathematica input statement to the Mathematica kernel. Finally, it translates back the Mathematica output statement into OpenMath XML encoding file.

This Phrasebook is written in JAVA. It works under the latest version of Mathematica (Mathematica 9.0 [12]). It includes a collection of JAVA classes of encoding and decoding methods between OpenMath and Mathematica based on the declaration of the corresponding CDs, respectively. In addition, we implemented JAVA methods that define the Mathematica calling functions with the string already specified by the user. As an example we assume that the user needs Mathematica to compute the addition of 1 and 2 such as `"1 + 2" "FullSimplify"`. `"FullSimplify"` in this example is the specific Mathematica function that a user calls from Mathematica. Moreover, this JAVA application includes a JAR file called **"om-lib.jar"** [27].

This JAR file represents the library of OpenMath. It is an Application Programming Interface (API) for reading, writing and creating OpenMath objects. This library comprises a set of packages that implement the basic OpenMath standard such as basic OpenMath Objects (e.g. integer, float and variable) and compound OpenMath objects (e.g application, binding, attribution and errors).

This phrasebook includes also the Mathematica service [11] which allows users to call remotely Mathematica kernel as a computational engine. This connection is established using J/Link library via TCP/IP protocol. The J/Link library is a developer's kit for JAVA used as a Wolfram Research's protocol for sending data and commands between JAVA programs and Mathematica kernel. The core of J/Link is just such a translation layer of the MathLink library which is implemented as a library of C-language. Figure A.4 shows the interface of the Mathematica service. The user needs to specify the host machine that contains Mathematica and the Mathematica path. For example, as described in Figure A.4, the host machine name is: `cosmic.encs.concordia.ca` and the path to Mathematica is: `C:\Program Files\Wolfram Research\Mathematica\8.0`.



hostname: cosmic.encs.concordia.ca

command: C:\Program Files\Wolfram Research\Mathematica\8.0

Figure 4.3: Mathematica Service

The OpenMath-Mathematica Phrasebook comprises of a large set of useful official and experimental CDs such as: `arith1`, `arith2`, `relation1`, etc. [18]. It does not include the recent ones such as `linalg4` [19] and `linalgeig1` [20] which describe the useful functions related to eigenvalues and eigenvectors functions. Thus, we added the implementation of the encoding/decoding methods of those CDs.

In the next subsection, we describe how we connect the OpenMath-Mathematica Phrasebook to the OCaml unit.

4.1.4 Connecting OCaml and OpenMath-Mathematica Phrasebook

In order to connect our OCaml part to the OpenMath-Mathematica Phrasebook, we generate a JAR file of the Phrasebook. Then, we launch it as a background process from OCaml file called **"MathematicaPhrasebook.ml"**.The code below starts by loading the **"unix.cma"** [28] library which is the Unix library written in OCaml that allows the communications between processes. Then, this file reads and prints line by line the returned result from Mathematica. Once the returned result is printed we stop the launched process.

```
#load "unix.cma";;
let call_mathematicaPhrasebook () =
   let oc_in, oc_out = Unix.open_process " java -jar
      MathematicaPhrasebook.jar"
in print_string (input_line oc_in) ;
print_string (input_line oc_in) ;
flush stdout ;
flush oc_out ;
Unix.close_process (oc_in, oc_out)
```

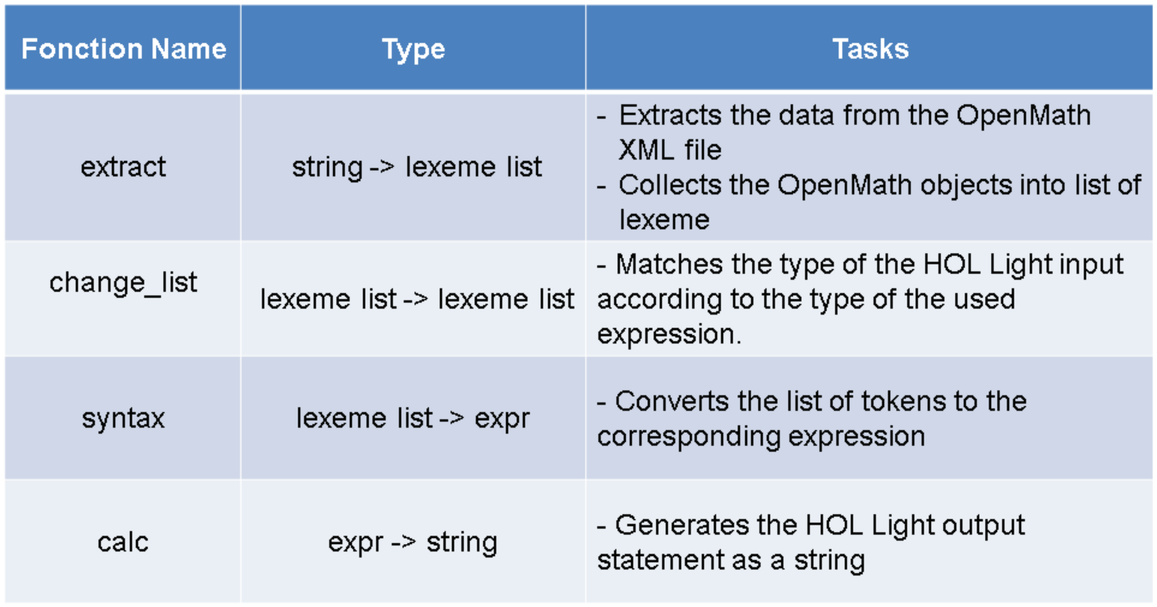
4.1.5 The Parser & Collector

The *"Parser & Collector"* translates the OpenMath object into the corresponding HOL Light expression in the relevant CDs, using the function `extract`. First, it reads OpenMath XML encoding file returned from the OpenMath-Mathematica Phrasebook. Second it loads the XML-Light library to extract the data from the XML tags. Then, it generates a preliminary lexeme list which is passed to the function `change_list` in order to check and match the correct type to each existing mathematical functions. This updated list is translated to the corresponding mathematical expression by the function `syntax`. Finally, the function `calc` returns the HOL Light output statment.

```
let call_collector() =
calc(syntax(change_list(extract file)))
```

Table 4.2 describes the different functionalities of the *"Parser & Collector"* module.

Table 4.2: Parser & Collector Internal Functions



Fonction Name	Type	Tasks
extract	string -> lexeme list	- Extracts the data from the OpenMath XML file - Collects the OpenMath objects into list of lexeme
change_list	lexeme list -> lexeme list	- Matches the type of the HOL Light input according to the type of the used expression.
syntax	lexeme list -> expr	- Converts the list of tokens to the corresponding expression
calc	expr -> string	- Generates the HOL Light output statement as a string

The XML-Light library [29] parses and prints OCaml statments to/from XML description. It provides functions that convert an XML document into an OCaml data structure, manipulates it, for example by modifying its content, and prints it back to an XML document.

The use of the XML-Light library is to extract data from the XML encoding OpenMath file as described below :

```
let rec extract_data acc = function
   | Xml.Element ("OMA",_, children) -> List (List.fold_left
      extract_data [] children) :: acc
   | Xml.Element ("OMBIND", _, children) -> List (List.fold_left
      extract_data [] children) :: acc
   | Xml.Element ("OMBVAR", _, children) -> acc
   | Xml.Element ("OMS", attr, _) -> let name = List.assoc "name"
      attr in (match name with
                  "pi" -> Str name :: acc
                  | "e" -> Str name :: acc
                  | "lambda" -> acc
                  | _ -> Symb name :: acc)
   | Xml.Element ("OMV", attr, _) -> Str (List.assoc "name" attr) ::
      acc
   | Xml.Element ("OMF", attr, _) -> Numf (float\_of\_string (List.
      assoc "dec" attr)) :: acc
   | Xml.Element ("OMI", _, [child]) -> Num (Xml.pcdata child) :: acc
   | _ -> acc
```

The code above explains how we extract the data of the mathematical expression from the XML OpenMath file. We distinguish two types of XML data structures: `XML.Element` and `XML.PCData`. Both of them denote an XML node which can be either **Element (tag-name, attributes, children)** or **PCData text**, which are the smallest entity of valid structures in an XML document.

For example, assume that we have the OpenMath XML encoding:

```
<OMA>
<OMS name="abs" cd="arith1"/>
<OMI> - 4 </OMI>
</OMA>
```

The XML data structure corresponding to the above encoding is given as:

```
Element(OMA,[], [Element ("OMS", [["name";"abs"]; ["cd";"arith1"
   ]],[]); Element ("OMI",[], [PCDATA "-4"])])
```

This encoding defines that the Element of the tag `OMA` does not have any attribute, but it has two children. The first child is the Element tagged by `OMS` and the attributes `name="abs"` and `cd="arith1"` without any children (empty list). The second child is tagged by `OMI` with a PCDATA as a child defined by `"-4"`, but without any attributes (empty list). In addition, we define the extraction of the elements of each existing tag in the OpenMath XML encoding. As a result, we get a list of lexical items existing in the returned result. These tokens correspond to the same grammar that we define in the *Parser & Splitter* (explained previously in Section 4.1.1.2). Once we get the OpenMath Object, we collect all the OpenMath objects and translate them into the HOL Light expression as understood in the corresponding CDs.

4.2 Applications

Our current implementation handles a large set of mathematical operations such as polynomial roots, factorization, evaluating and solving arithmetic expressions and matrix operations. In our experiments, we conducted 20 applications and the average execution time is around 3 seconds. In the following subsection, we will show some of the running applications such as the evaluating and solving of matrix operations. We conclude this section by presenting an application which verifies the boundary condition of an optical interface using Mathematica.

4.2.1 Matrix Operations

In this subsection, first we give a brief description how to define a matrix 2x2 in HOL Light. Then, we show some running examples such as arithmetic operations of matrices 2x2, evaluating the determinant or computation of the eigenvalues and the eigenvectors of a matrix 2x2.

4.2.1.1 Structure of Matrix 2x2

We define a matrix 2x2 in HOL Light syntax as follows : `HOL Light Symbol`

```
mat2x2 a b c d
```

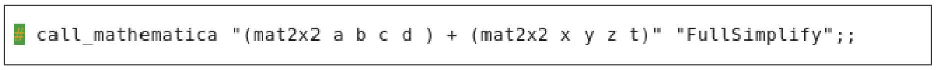
4.2.1.2 Arithmetic Operation of Matrices 2x2

Our implementation handles arithmetic operations between matrices such as addition, subtraction, multiplication, division and power functions. As an example, we show the addition of two matrices below :

Consider the mathematical expression :

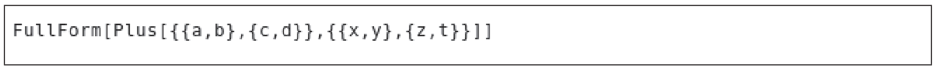
$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} + \begin{pmatrix} x & y \\ z & t \end{pmatrix}$$

We call Mathematica with the following HOL Light statement :



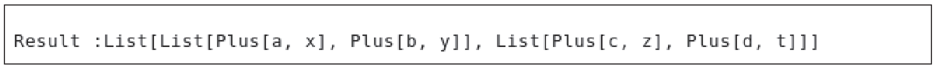
```
# call_mathematica "(mat2x2 a b c d ) + (mat2x2 x y z t)" "FullSimplify";;
```

The Mathematica input statement is :



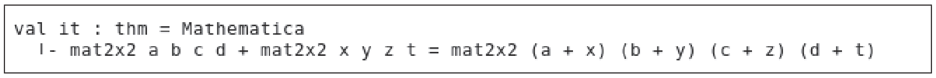
```
FullForm[Plus[{{a,b},{c,d}},{{x,y},{z,t}}]]
```

After Computation, we get the following result in Mathematica syntax :



```
Result :List[List[Plus[a, x], Plus[b, y]], List[Plus[c, z], Plus[d, t]]]
```

Finally, the returned theorem is represented as follows in HOL Light :



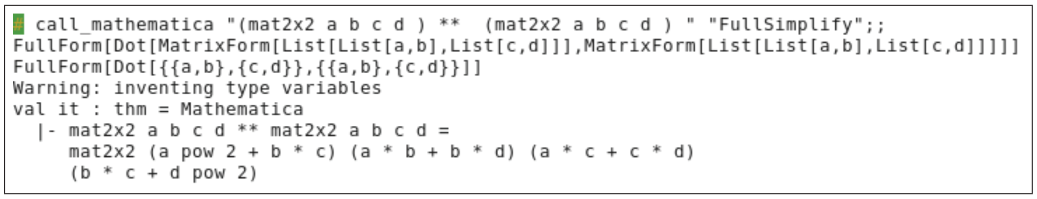
```
val it : thm = Mathematica
  |- mat2x2 a b c d + mat2x2 x y z t = mat2x2 (a + x) (b + y) (c + z) (d + t)
```

which represents the following mathematical expression :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} + \begin{pmatrix} x & y \\ z & t \end{pmatrix} = \begin{pmatrix} a+x & b+y \\ c+z & d+t \end{pmatrix}$$

The same process is applied for the multiplication between two matrices as described in Figure 4.4 :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} x & y \\ z & t \end{pmatrix} = \begin{pmatrix} a*x+b*z & b*t+a*y \\ c*x+d*z & d*t+c*y \end{pmatrix}$$

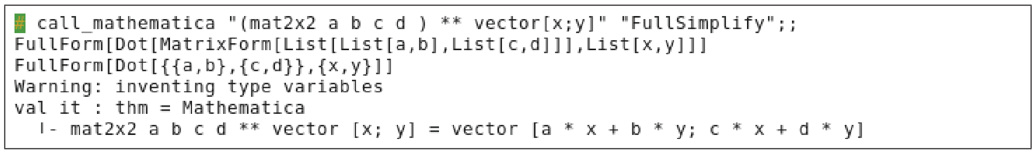


```
call_mathematica "(mat2x2 a b c d ) **  (mat2x2 a b c d ) " "FullSimplify";;
FullForm[Dot[MatrixForm[List[List[a,b],List[c,d]]],MatrixForm[List[List[a,b],List[c,d]]]]]
FullForm[Dot[{{a,b},{c,d}},{{a,b},{c,d}}]]
Warning: inventing type variables
val it : thm = Mathematica
  |- mat2x2 a b c d ** mat2x2 a b c d =
     mat2x2 (a pow 2 + b * c) (a * b + b * d) (a * c + c * d)
     (b * c + d pow 2)
```

Figure 4.4: Multiplication of Two 2x2 Matrices

Another example shows the multiplication of a matrix and a vector defined by the symbol ** in HOL Light, as described in Figure 4.5 :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a*x+b*y \\ c*x+d*y \end{pmatrix}$$



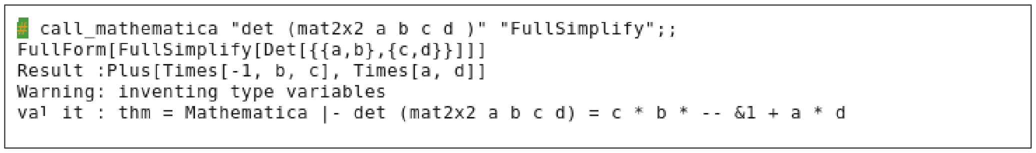
```
# call_mathematica "(mat2x2 a b c d ) ** vector[x;y]" "FullSimplify";;
FullForm[Dot[MatrixForm[List[List[a,b],List[c,d]]],List[x,y]]]
FullForm[Dot[{{a,b},{c,d}},{x,y}]]
Warning: inventing type variables
val it : thm = Mathematica
  |- mat2x2 a b c d ** vector [x; y] = vector [a * x + b * y; c * x + d * y]
```

Figure 4.5: Multiplication of a 2x2 Matrix and a Vector

4.2.1.3 Determinant

The computation of the determinant of a matrix 2x2 is described in Figure 4.6. A determinant is a function of a square matrix that reduces it to a real number. The determinant of a matrix `A` is denoted by |A| or `det(A)`. If `A` consists of one element `a`, then |`A`|=`a`. If `A` is a 2x2 matrix, then

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$



```
# call_mathematica "det (mat2x2 a b c d )" "FullSimplify";;
FullForm[FullSimplify[Det[{{a,b},{c,d}}]]]
Result :Plus[Times[-1, b, c], Times[a, d]]
Warning: inventing type variables
val it : thm = Mathematica |- det (mat2x2 a b c d) = c * b * -- &1 + a * d
```

Figure 4.6: Determinant of a Matrix 2x2

The execution time of the computation of the determinant of a matrix 2x2 took 2.1 seconds.

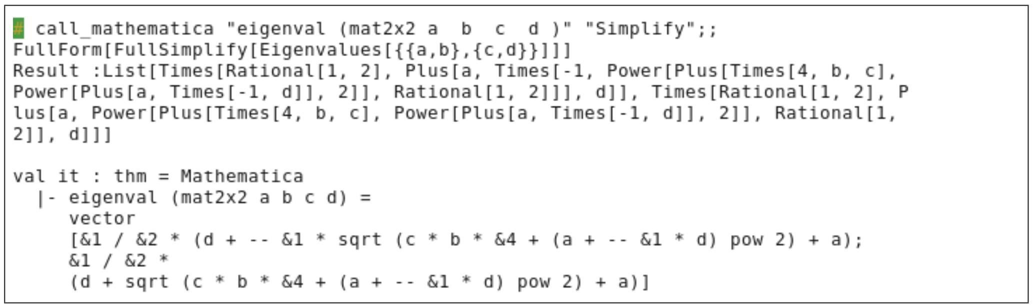
4.2.1.4 Eigenvectors and Eigenvalues

Our implementation also handles the computation of the eigenvalues and eigenvectors of a matrix 2x2 as described in Figures 4.6 and 4.8, respectively.

An eigenvector is a nonzero vector that satisfies the equation

$$A\vec{v} = \lambda\vec{v}$$

where `A` is a square matrix, λ is a scalar, and $\vec{v}$ is the eigenvector. λ is called an eigenvalue.

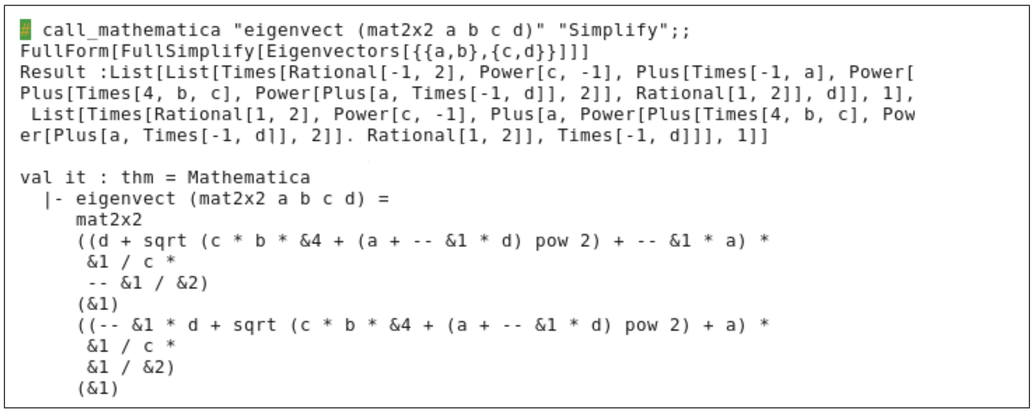


```
# call_mathematica "eigenval (mat2x2 a  b  c  d )" "Simplify";;
FullForm[FullSimplify[Eigenvalues[{{a,b},{c,d}}]]]
Result :List[Times[Rational[1, 2], Plus[a, Times[-1, Power[Plus[Times[4, b, c],
Power[Plus[a, Times[-1, d]], 2]], Rational[1, 2]]], d]], Times[Rational[1, 2], P
lus[a, Power[Plus[Times[4, b, c], Power[Plus[a, Times[-1, d]], 2]], Rational[1,
2]], d]]]

val it : thm = Mathematica
  |- eigenval (mat2x2 a b c d) =
     vector
     [&1 / &2 * (d + -- &1 * sqrt (c * b * &4 + (a + -- &1 * d) pow 2) + a);
     &1 / &2 *
     (d + sqrt (c * b * &4 + (a + -- &1 * d) pow 2) + a)]
```

Figure 4.7: Eigenvalues of a Matrix 2x2

The execution time of the computation of eigenvalues took 2.3 seconds.



```
# call_mathematica "eigenvect (mat2x2 a b c d)" "Simplify";;
FullForm[FullSimplify[Eigenvectors[{{a,b},{c,d}}]]]
Result :List[List[Times[Rational[-1, 2], Power[c, -1], Plus[Times[-1, a], Power[
Plus[Times[4, b, c], Power[Plus[a, Times[-1, d]], 2]], Rational[1, 2]], d]], 1],
 List[Times[Rational[1, 2], Power[c, -1], Plus[a, Power[Plus[Times[4, b, c], Pow
er[Plus[a, Times[-1, d]], 2]]. Rational[1, 2]], Times[-1, d]]], 1]]

val it : thm = Mathematica
  |- eigenvect (mat2x2 a b c d) =
     mat2x2
     ((d + sqrt (c * b * &4 + (a + -- &1 * d) pow 2) + -- &1 * a) *
      &1 / c *
      -- &1 / &2)
     (&1)
     ((-- &1 * d + sqrt (c * b * &4 + (a + -- &1 * d) pow 2) + a) *
      &1 / c *
      &1 / &2)
     (&1)
```

Figure 4.8: Eigenvectors of a Matrix 2x2

The execution time of the computation of eigenvalues took 2.6 seconds.

4.2.2 Boundary Condition of an Optical Interface

To show the effectiveness of our approach and implementation to connect HOL Light and Mathematica, we present in this section the example of a boundary condition computing of an optical interface [56] described with electromagnetic field. Figure 4.9 shows the system of interface that includes the interface `i` and electromagnetic fields $EMF^{(i)}$, $EMF^{(r)}$, and $EMF^{(t)}$.

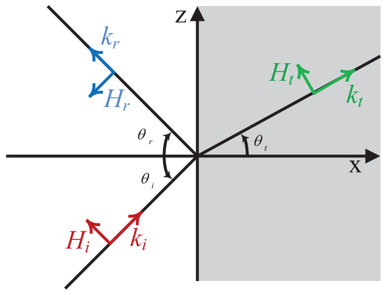


Figure 4.9: The System of Interface

Figure 4.9 is the yz-plane; our objective is to validate that the electromagnetic fields $EMF^{(i)}$, $EMF^{(r)}$, and $EMF^{(t)}$ satisfy the boundary conditions. This can be formally described as follows:

$\forall \mathtt{r}\ \mathtt{t}.$

$\mathtt{let}\ \theta_t = \arcsin(\frac{n_1}{n_2}\sin\theta_i)\ \mathtt{in}\ \mathtt{let}\ \theta_r = \theta_i\ \mathtt{in}$

$\mathtt{let}\ r_a = (\frac{n_2\cos\theta_i - n_1\cos\theta_t}{n_2\cos\theta_i + n_1\cos\theta_t})a\ \mathtt{in}\ \ \mathtt{let}\ t_a = (\frac{2n_2\cos\theta_i}{n_2\cos\theta_i + n_1\cos\theta_t})a\ \mathtt{in}$

$\mathtt{let}\ k^{(i)} = k_0 n_1[\cos\theta_i;\ 0;\ \sin\theta_i]\ \mathtt{in}$

$\mathtt{let}\ k^{(r)} = k_0 n_1[-\cos\theta_r;\ 0;\ \sin\theta_r]\ \mathtt{in}$

$\mathtt{let}\ k^{(t)} = k_0 n_2[\cos\theta_t;\ 0;\ \sin\theta_t]\ \mathtt{in}$

$\mathtt{let}\ E^{(i)} = [0;\ ae^{-j\ (k^{(i)}.\ r-\omega t)};0]\ \mathtt{in}$

$\mathtt{let}\ E^{(r)} = [0;\ r_a e^{-j\ (k^{(r)}.\ r-\omega t)};0]\ \mathtt{in}$

$\mathtt{let}\ E^{(t)} = [0;\ t_a e^{-j\ (k^{(t)}.\ r-\omega t)};0]\ \mathtt{in}$

$\mathtt{let}\ H^{(i)} = [a(\frac{n_1}{eta_0})\cos\theta_i\ e^{-j(k^{(i)}.\ r-\omega t)};\ 0;\ -a(\frac{n_1}{eta_0})\sin\theta_i\ e^{-j(k^{(i)}.\ r-\omega t)}]\ \mathtt{in}$

$\mathtt{let}\ H^{(r)} = [-r_a(\frac{n_1}{eta_0})\cos\theta_r\ e^{-j(k^{(r)}.\ r-\omega t)};\ 0;\ -r_a(\frac{n_1}{eta_0})\sin\theta_r\ e^{-j(k^{(i)}.\ r-\omega t)}]\ \mathtt{in}$

$\mathtt{let}\ H^{(t)} = [t_a(\frac{n_2}{eta_0})\cos\theta_t\ e^{-j(k^{(t)}.\ r-\omega t)};\ 0;\ -t_a(\frac{n_2}{eta_0})\sin\theta_t\ e^{-j(k^{(t)}.\ r-\omega t)}]\ \mathtt{in}$

$\mathtt{let}\ EMF^{(i)} = \mathtt{plane\_wave}\ k^{(i)}\ \omega\ E^{(i)}\ H^{(i)}\ \mathtt{in}$

$\mathtt{let}\ EMF^{(r)} = \mathtt{plane\_wave}\ k^{(r)}\ \omega\ E^{(r)}\ H^{(r)}\ \mathtt{in}$

$\mathtt{let}\ EMF^{(t)} = \mathtt{plane\_wave}\ k^{(t)}\ \omega\ E^{(t)}\ H^{(t)}\ \mathtt{in}$

$\forall \mathtt{pt}.\ \mathtt{pt\ IN\ (plane\_of\_interface\ i)} \wedge (\mathtt{pt}\$1 = 0) \Rightarrow$

$\mathtt{boundary\_conditions}\ (EMF^{(i)} + EMF^{(r)})\ EMF^{(t)}\ (\mathtt{plane\_of\_interface\ i})\ \mathtt{r}\ \mathtt{t}$

Figure 4.10: Goal of the Boundary Conditions of the System of Interface in Figure 4.9 [56].

After simplifying the goal in Figure 4.10, we derive the expression shown in the Figure 4.11, which we send to Mathematica.

$$
\begin{aligned}
&[1;0;0]\ \mathtt{ccross}\\
&([0\,;\,ae^{-j(k_0 n_1 z\sin\theta_i-\omega t)};0]\ +\\
&\quad[0\,;\,\frac{n_2\cos\theta_i-n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))}{n_2\cos\theta_i+n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))}ae^{-j(k_0 n_1 z\sin\theta_i-\omega t)};\ 0])\ =\\
&\quad[1;0;0]\ \mathtt{ccross}\\
&\quad[0\,;\,\frac{2n_2\cos\theta_i}{n_2\cos\theta_i+n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))}a\,e^{-j(k_0 n_1 z\sin(\arcsin(\frac{n_1}{n_2}\sin\theta_i))-\omega t)};\ 0]\\
&\quad\wedge\\
&\quad[1;0;0]\ \mathtt{ccross}\\
&\quad([a\frac{n_1}{\eta_0}\cos\theta_i e^{-j(k_0 n_1 z\sin\theta_i-\omega t)};0\,;-a\frac{n_1}{\eta_0}\sin\theta_i e^{-j(k_0 n_1 z\sin\theta_i-\omega t)}]\ +\\
&\quad[-\frac{n_2\cos\theta_i-n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))}{n_2\cos\theta_i+n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))}a\frac{n_1}{\eta_0}\cos\theta_i e^{-j(k_0 n_1 z\sin\theta_i-\omega t)};\ 0;\\
&\quad-\frac{n_2\cos\theta_i-n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))}{n_2\cos\theta_i+n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))}a\frac{n_1}{\eta_0}\sin\theta_i e^{-j(k_0 n_1 z\sin\theta_i-\omega t)}]\ =\\
&\quad[1;0;0]\mathtt{ccross}\\
&\quad[\frac{2n_2\cos\theta_i)}{n_2\cos\theta_i+n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))}a\frac{n_2}{\eta_0}\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i)e^{-j(k_0 n_1 z\sin(\arcsin(\frac{n_1}{n_2}\sin\theta_i))-\omega t)};\ 0;\\
&\quad-\frac{2n_2\cos\theta_i}{n_2\cos\theta_i+n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))}a\frac{n_2}{\eta_0}\sin(\arcsin(\frac{n_1}{n_2}\sin\theta_i))e^{-j(k_0 n_1 z\sin(\arcsin(\frac{n_1}{n_2}\sin\theta_i))-\omega t)}])
\end{aligned}
$$

Figure 4.11: The Expression Sent to Mathematica to be Simplified/Verified [56].

In the Goal of Figure 4.11, the predicate boundary_conditions is reduced to the cross product between the normal to the interface (i.e., [1 0 0]) and summation of electric fields or magnetic fields at the interface. Now, we can validate the Goal of Figure 4.11 by calling Mathematica which returns the following theorem:

```
Input:
```

#call_mathematica Statement of Goal of the Figure 4.11 “FullSimplify”

```
Output:
```

$$
\begin{aligned}
&\mathtt{val\ it : thm} = \vdash \\
&[1;0;0]\ \mathtt{ccross} \\
&([0\ ; a e^{-j(k_0 n_1 z \sin\theta_i - \omega t)}; 0] + [0\ ; \frac{n_2\cos\theta_i - n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))}{n_2\cos\theta_i + n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))} a e^{-j(k_0 n_1 z \sin\theta_i - \omega t)};\ 0]) = \\
&[1;0;0]\ \mathtt{ccross}\ [0\ ; \frac{2n_2\cos\theta_i}{n_2\cos\theta_i + n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))} a\, e^{-j(k_0 n_1 z \sin(\arcsin(\frac{n_1}{n_2}\sin\theta_i)) - \omega t)};\ 0] \\
&\wedge \\
&[1;0;0]\ \mathtt{ccross} \\
&([a\frac{n_1}{\eta_0}\cos\theta_i e^{-j(k_0 n_1 z \sin\theta_i - \omega t)}; 0\ ; -a\frac{n_1}{\eta_0}\sin\theta_i e^{-j(k_0 n_1 z \sin\theta_i - \omega t)}] + \\
&[-\frac{n_2\cos\theta_i - n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))}{n_2\cos\theta_i + n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))} a\frac{n_1}{\eta_0}\cos\theta_i e^{-j(k_0 n_1 z \sin\theta_i - \omega t)};\ 0; \\
&\ -\frac{n_2\cos\theta_i - n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))}{n_2\cos\theta_i + n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))} a\frac{n_1}{\eta_0}\sin\theta_i e^{-j(k_0 n_1 z \sin\theta_i - \omega t)}] = \\
&[1;0;0]\mathtt{ccross} \\
&[\frac{2n_2\cos\theta_i)}{n_2\cos\theta_i + n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))} a\frac{n_2}{\eta_0}\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i) e^{-j(k_0 n_1 z \sin(\arcsin(\frac{n_1}{n_2}\sin\theta_i)) - \omega t)};\ 0; \\
&\ -\frac{2n_2\cos\theta_i}{n_2\cos\theta_i + n_1\cos(\arcsin(\frac{n_1}{n_2}\sin\theta_i))} a\frac{n_2}{\eta_0}\sin(\arcsin(\frac{n_1}{n_2}\sin\theta_i)) e^{-j(k_0 n_1 z \sin(\arcsin(\frac{n_1}{n_2}\sin\theta_i)) - \omega t)}])
\end{aligned}
$$

The total verification time for this proof on a computer (Intel i7, 2.8GHz CPU with 32GB RAM) running OS Oracle Linux 6.3 was about 2.87 seconds, which includes the time to invoke Mathematica and get its feedback. This example illustrates the usefulness of the CAS link that greatly facilitates the interactive theorem proving process by automatically verifying some of the proof goals that would have required hours of user guidance if verified by theorem proving alone.

4.3 Summary

In this chapter, we first presented an overview of the implementation of our tool. Second, we described the common data types that we defined. Then, we gave details about the implementation of the *“Parser & Splitter”* and the *“Parser & Collector”* modules. We described also the connection between the HOL Light translator and OpenMath-Mathematica Phrasebook. Finally, we showed the effectiveness of our tool by presenting some running examples and applications. In Appendix A, we will describe the installation guide of our tool. Then, we present the process of a running example.

Chapter 5

Conclusion and Future work

5.1 Conclusion

In this thesis we presented a link between HOL Light and Mathematica through OpenMath. This link is a prototype aiming at building a general framework that allows the integration of algebraic computation and deductive reasoning in solving mathematical problems. In this linkage, OpenMath represents a common mathematical knowledge representation that can be exchanged between CASs and TPSs. Moreover, our work sharpen the complementarity of the use of the algebraic computation and the deductive reasoning in solving sophisticated mathematical problems. Although both of TPSs and CASs perform symbolic computation, each system has it own singularity. TPSs provide an accurate result while CASs are efficient for computation.

Our implementation mainly consists on developing a HOL Light translator which converts HOL Light expressions into OpenMath encoding and vice-versa. Our HOL Light translator relies on its own grammar that is implemented without modifying the internal structure of HOL Light. Thanks to this translator, HOL Light users can access Mathematica's kernel using the Phrasebook OpenMath-Mathematica proposed by Caprotti [40]. We implemented an OCaml unit which transforms the HOL

Light statement into a corresponding OpenMath object as understood by means of the CD. In addition, we implemented an OCaml unit which translates back the OpenMath object into the specification HOL Light symbols in the relevant CDs.

In addition, we illustrate the usefulness of our approach by presenting several running examples such as calling Mathematica in order to solve or evaluate a non-closed form solution such as arithmetic or polynomial manipulations or matrix operations. These examples emphasize not only the benefits of computing such mathematica expressions within HOL Light but also the efficient performance of our tool in terms of execution time. In fact, running those examples such as computing the eigenvalues of general 2x2 matrix takes only few seconds to call Mathematica and get its feedback.

Finally, our experimental prototype called *HolMatica* is implemented in a way that we can easily adapt it to any other CAS or TPS that supports OpenMath. All our codes and files can be downloaded on the web site `http://hvg.ece.concordia.ca/research/tools/holmatica/`. The installation guide and a description of a running example are described in Appendix A.

5.2 Future Work

The linkage between Mathematica as a computer algebra system and HOL Light as a theorem prover system, presented in this thesis, is an experimental prototype. Diverse future work directions can be performed building upon this work.

- Currently, we can handle some useful operations such as eigenvalues computation, evaluating and solving arithmetic expressions, polynomial roots and factorization. We can extend our work to target more complex operations such as complex polynomial expression, partial differentiation, etc. Moreover, our current implementation handles only real numbers, we can extend it to handle complex field numbers and implement the mathematical operations related to this field such as the argument of the conjugate of a complex number. In this case, we should check the CD related to these mathematical expression and add them in the parsing and mapping function in the *Parser & Splitter*. Also we should add the encoding/decoding JAVA methods in the Phrasebook if they are not included.

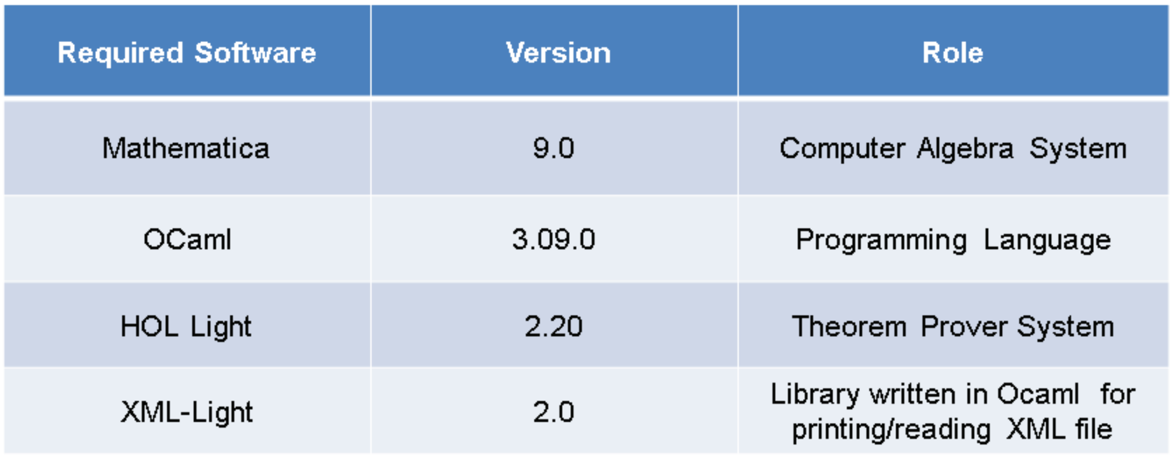
- One of the limitations of the proposed methodology is that we work with the Mathematica service. This service allows users to access remotely to Mathematica via TCP/IP protocol. The limitation of this service is that the user need to specify the host name and the path to Mathematica which may be not available in sometimes. To overcome these issues, we propose in the future to implement a web service which allows the user to access directly Mathematica via Internet which specify any additional configurations.

- In our current system, we implement a procedure that automates the proof steps of the returned result from Mathematica. The returned result from Mathematica is represented as HOL Light theorem or a sub-goal. However, this automatization works only when the returned result is a sub-goal and when the sub-goal includes only simple arithmetic operations such as addition, subtraction and multiplication. As a future direction for this problem, we can develop HOL Light function that proves the simplified result to ensure the soundness of the Mathematica returned expression. To do so, we can record the simplification algorithm that Mathematica applied on the mathematical expression and build the proof steps based on that.

Appendix A

A.1 Required Software

The setup of our tool involves installing the following required software tools and libraries described below in Table A.1.



Required Software	Version	Role
Mathematica	9.0	Computer Algebra System
OCaml	3.09.0	Programming Language
HOL Light	2.20	Theorem Prover System
XML-Light	2.0	Library written in Ocaml for printing/reading XML file

Table A.1: The Required Software for the Implementation

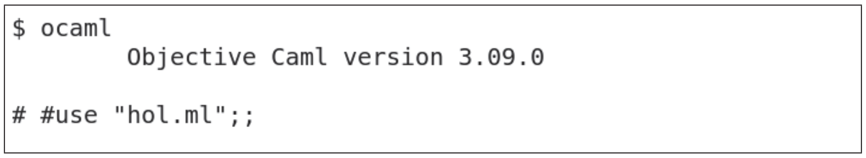
Before we start the tool, we must first set up our development environment. The connection that we implemented involves that we access to HOL Light, which is a library built on top of OCaml, and Mathematica. In addition, for the internal implementation we need to install the XML-Light library which manipulates XML files. In the next section, we will see how to install the tool.

A.2 Tool Installation

The setting up of the tool involves the installation of the required software mentioned in Section A.1. All of our code and files are public and can be downloaded from the web site `http://hvg.ece.concordia.ca/research/tools/holmatica/`. In the following we present the steps to install and run an example :

– First, extract the compressed file : ”HolLight-Mathematica.zip” in the HOL Light directory.

– Second, load the HOL Light library by calling the “hol.ml” file as described in Figure A.1.

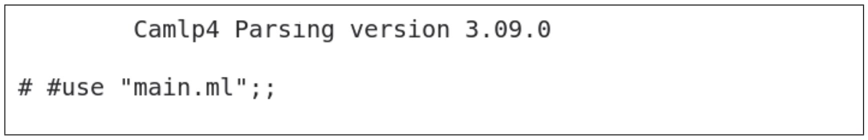


```
$ ocaml
        Objective Caml version 3.09.0

# #use "hol.ml";;
```

Figure A.1: HOL Light Session

– Then, load the principle OCaml library of the tool by calling the “main.ml” file as described in Figure A.2.

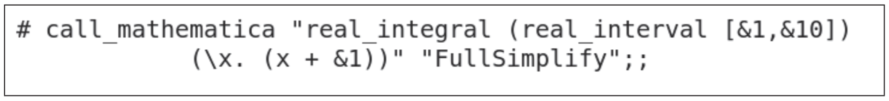


```
        Camlp4 Parsıng version 3.09.0

# #use "main.ml";;
```

Figure A.2: Main Program Function

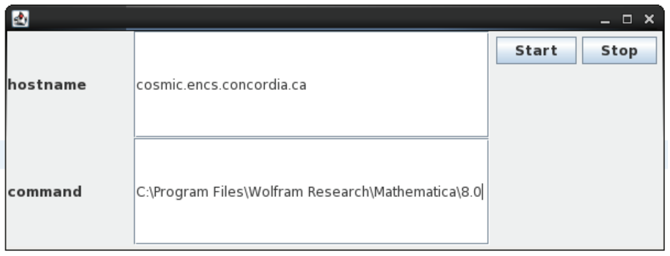
– Type the HOL Light input statement which calls the main function “call_mathematica” that accepts the HOL Light expression and the Mathematica function.



```
# call_mathematica "real_integral (real_interval [&1,&10])
           (\x. (x + &1))" "FullSimplify";;
```

Figure A.3: HOL Light Expression of Computing Real Integral

– The Mathematica service dialogue window appears and the user specifies the name of the host machine where Mathematica is installed and the path to the Mathematica kernel as shown in Figure A.4 shows this step:



hostname: cosmic.encs.concordia.ca

command: C:\Program Files\Wolfram Research\Mathematica\8.0

Figure A.4: Mathematica Service

– After launching the Mathematica service, we get the computed result in the console as described in Figure A.5

```
# call_mathematica "real_integral (real_interval [&1,&10])
        (\x. (x + &1))" "FullSimplify";;

val it : thm = Mathematica
  |- real_integral (real_interval [&1,&10])
                      (\x. x + &1) = &117 / &2
```

Figure A.5: HOL Light Result after Computation by Mathematica

Bibliography

[1] Axiom The Scientific Computation System . `http://www.axiom-developer.org/`, 2014.

[2] Domains of Use of Mathematica. `http://www.wolfram.com/solutions/`, 2014.

[3] FullSimplify. `http://reference.wolfram.com/mathematica/ref/FullSimplify.html`, 2014.

[4] GAP System for Computational Discrete Algebra. `http://www.gap-system.org/`, 2014.

[5] HOL4. `http://hol.sourceforge.net/`, 2014.

[6] Introduction to MathLink . `http://reference.wolfram.com/mathematica/tutorial/IntroductionToMathLink.html`, 2014.

[7] Isabelle. `http://isabelle.in.tum.de/`, 2014.

[8] KeYmaera: A Hybrid Theorem Prover for Hybrid Systems. `http://symbolaris.com/info/KeYmaera.html`, 2014.

[9] LEGO Proof Development System. `http://www.lfcs.inf.ed.ac.uk/reports/92/ECS-LFCS-92-211/`, 2014.

[10] Maple. `http://www.maplesoft.com/`, 2014.

[11] MathDox. http://mathdox.org/new-web/index.html, 2014.

[12] Mathematica. http://www.wolfram.com/, 2014.

[13] MathLink API. http://reference.wolfram.com/mathematica/guide/MathLinkAPI.html, 2014.

[14] MathML - World Wide Web Consortium. http://www.w3.org/Math/, 2014.

[15] MATLAB. http://www.mathworks.com/products/matlab/, 2014.

[16] Message to info-HOL Mailing List. ftp://ftp.cl.cam.ac.uk/hvg/info-hol-archive/09xx/0972, 2014.

[17] MuPAD. http://www.mathworks.com/discovery/mupad.html, 2014.

[18] OpenMath Content Dictionaries. http://www.openmath.org/cd/index.html, 2014.

[19] OpenMath Content Dictionary: linalg4. http://www.openmath.org/cd/linalg4.xhtml, 2014.

[20] OpenMath Content Dictionary: linalgeig1. http://www.win.tue.nl/~amc/oz/om/cds/linalgeig1.xml, 2014.

[21] Oriented Object Caml. http://ocaml.org/index.fr.html, 2014.

[22] Overview of OpenMath. http://www.openmath.org/overview/index.html, 2014.

[23] PVS Specification and Verification System. http://pvs.csl.sri.com/, 2014.

[24] REDUCE Computer Algebra System. http://reduce-algebra.com/, 2014.

[25] The Coq Proof Assistant. http://coq.inria.fr/, 2014.

[26] The HOL Light Theorem Prover. `http://www.cl.cam.ac.uk/\~jrh13/hol-light/`, 2014.

[27] The JAVA OpenMath Library. `http://team.polylab.sfu.ca/openmath0.5/lib/`, 2014.

[28] The Unix Library: Unix System Calls Available to OCaml Programs. `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Unix.html`, 2014.

[29] The XML-LIGHT Library. `http://tech.motion-twin.com/xmllight.html`, 2014.

[30] Verification of Object-Oriented Software: The KeY Approach. `http://www.key-project.org/thebook/`, 2014.

[31] A. Bauer and E. Clarke and X. Zhao. Analytica - An Experiment in Combining Theorem Proving and Symbolic Computation. *Journal of Automated Reasoning*, 21(3):295–325, 1998.

[32] A. Dolzmann and T. Sturm. REDLOG: Computer Algebra Meets Computer Logic. *Association for Computing Machinery Special Interest Group on Symbolic and Algebraic Manipulation Bulletin*, 31(2):2–9, 1997.

[33] A. Adams, M. Dunstan, H. Gottliebsen, T. Kelsey, U. Martin, and S. Owre. Computer Algebra Meets Automated Theorem Proving: Integrating Maple and PVS. In *Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 27–42. Springer-Verlag, 2001.

[34] P. B. Andrews, M. Bishop, S. Issar, D. Nesmith, F. Pfenning, and H. Xi. TPS: An Interactive and Automatic Tool for Proving Theorems of Type Theory. In *HUG*, volume 780 of *Lecture Notes in Computer Science*, pages 366–370. Springer, 1993.

[35] C. Ballarin, K. Homann, and J. Calmet. Theorems and Algorithms: An Interface between Isabelle and Maple. In *International Symposium on Symbolic and Algebraic Computation*, pages 150–157. ACM, 1995.

[36] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.

[37] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A Survey of the Theorema project. In *International Symposium on Symbolic and Algebraic Computation*, pages 384–391. Association for Computing Machinery Press, 1997.

[38] O. Caprotti and A. M. Cohen. Connecting Proof Checkers and Computer Algebra Using Openmath. In *Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 109–112. Springer, 1999.

[39] O. Caprotti and A. M. Cohen. Integrating computational and deduction systems using OpenMath. *Electronic Notes in Theoretical Computer Science*, 23(3):469–480, 1999.

[40] O. Caprotti, A. M. Cohen, and M. Riem. JAVA Phrasebooks for Computer Algebra and Automated Deduction. *Special Interest Group on Symbolic and Algebraic Manipulation Bulltin*, 34:33–37, 2000.

[41] J. Carette, W. M.Farmer, F. Jeremic, V. Maccio, R. O'Connor, and Q. M. Tran. The MathScheme Library: Some Preliminary Experiments. *Computing Research Repository*, abs/1106.1862, 2011.

[42] E. Clarke and X. Zhao. Analytica - A Theorem Prover for Mathematica. In *The Mathematica Journal*, pages 761–765. Springer-Verlag, 1993.

[43] S. Dalmas, M. Gatano, and S. M. Watt. An OpenMath 1.0 Implementation. In *International Symposium on Symbolic and Algebraic Computation*, pages 241–248. ACM, 1997.

[44] A. Grozin. *Computer Algebra Systems*. Springer, 2014.

[45] J. Y. Halpern and M. Y. Vardi. Model Checking vs. Theorem Proving: A Manifesto. In *Knowledge Representation*, pages 325–334. Morgan Kaufmann, 1991.

[46] R. Harper. *Programming in Standard ML*. 1998.

[47] J. Harrison. Theorem Proving for Verification (Invited Tutorial). In *Computer-Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 11–18. Springer, 2008.

[48] J. Davenport. *On Writing OpenMath Content Dictionaries*. OpenMath Esprit, 2002.

[49] J. Harrison and L. Théry. A Skeptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.

[50] C. Kaliszyk. *Correctness and Availability. Building Computer Algebra on top of Proof Assistants and Making Proof Assistants available over the Web.* PhD thesis, Radboud University Nijmegen, Netherlands, 2009.

[51] C. Kaliszyk and F. Wiedijk. Certified Computer Algebra on Top of an Interactive Theorem Prover. In *Calculemus/Mathematical Knowledge Management*, pages 94–105, 2007.

[52] Mathscheme. `http://www.cas.mcmaster.ca/research/mathscheme/`, 2014.

[53] O. Caprotti and A. M. Cohen. On the Role of OpenMath in Interactive Mathematical Documents. *Journal of Symbolic Computation*, 32(4):351–364, 2001.

[54] O. Caprotti and D. Carlisle. OpenMath and MathML: Semantic Markup for Mathematics. *Crossroads*, 6(2):11–14, 1999.

[55] A. Platzer and J. D. Quesel. KeYmaera: A Hybrid Theorem Prover for Hybrid Systems. In *International Journal of Computing Academic Research*, volume 5195 of *Lecture Notes in Computer Science*, pages 171–178. Springer, 2008.

[56] S. K. Afshar and U. Siddique and M. Y. Mahmoud and V. Aravantinos and O. Seddiki and O. Hasan and S. Tahar. Formal Analysis of Optical Systems. *Mathematics in Computer Science*, 8(1):39–70, 2014.

[57] J. van Benthem and K. Doets. *Higher-order logic*. Springer, 2001.

[58] J. Xu. *Mei : A Module System for Mechanized Mathematics Systems*. PhD thesis, 2008.