FORMAL COMPUTATIONS AND METHODS

by

Alexey Solovyev

B.S., St. Petersburg State University, St. Petersburg, Russia, 2007

Submitted to the Graduate Faculty of

the Kenneth P. Dietrich School of Arts and Sciences

in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2012

UNIVERSITY OF PITTSBURGH

KENNETH P. DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Alexey Solovyev

It was defended on

November 28, 2012

and approved by

Thomas C. Hales, Mellon Professor, Department of Mathematics, University of Pittsburgh

Gregory M. Constantine, Professor, Department of Mathematics, University of Pittsburgh

Bogdan Ion, Associate Professor, Department of Mathematics, University of Pittsburgh

Jeremy Avigad, Professor, Department of Philosophy, Carnegie Mellon University

Dissertation Director: Thomas C. Hales, Mellon Professor, Department of Mathematics, University of Pittsburgh

FORMAL COMPUTATIONS AND METHODS

Alexey Solovyev, PhD

University of Pittsburgh, 2012

We present formal verification methods and procedures for finding bounds of linear programs and proving nonlinear inequalities. An efficient implementation of formal arithmetic computations is also described. Our work is an integral part of the Flyspeck project (a formal proof of the Kepler conjecture) and we show how developed formal procedures solve formal computational problems in this project. We also introduce our implementation of SSReflect language (originally developed by G. Gonthier in Coq) in HOL Light.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

There are many people without whom this work would be not possible. I wish to thank Jeremy Avigad, Gregory Constantine, Joyeeta Dutta-Moscato, François Garillot, Georges Gonthier, Piotr Hajłasz, Thomas Hales, John Harrison, Bogdan Ion, Assia Mahboubi, Qi Mi, Russell O'Connor, Enrico Tassi, Yoram Vodovotz, Molly Williams, Leming Zhou, Cordelia Ziraldo, and Roland Zumkeller.

I appreciate the financial support of my work from the NSF, the Benter Foundation, and the University of Pittsburgh.

1.0 INTRODUCTION

This dissertation presents several tools and methods for doing formal computations and producing formal proofs in the HOL Light proof assistant [Har10, Har]. Our work is an integral part of the Flyspeck project by Thomas Hales [Hal12b, Hal06a]. The goal of this project is a formal proof of the Kepler conjecture [Hal12a, Hal06b, HHM^{+}09] and the primary formalization language is HOL Light.

The dissertation is organized as follows. Section 1.1 describes the Flyspeck project. Sections 1.2 and 1.3 briefly introduce our main formalization tool HOL Light and notions of formal proofs and computations. Section 1.4 presents main results of this work. Chapter 2 describes our implementation of an efficient formal arithmetic in HOL Light. Results of this chapter are used in Chapters 3 and 4 where general formal verification procedures for linear and nonlinear inequalities are presented. Also, these two chapters describe how general procedures can be applied to verify corresponding Flyspeck problems. Chapter 5 introduces a special mode for creating formal proofs in HOL Light which has been inspired by SSReflect formal proof language by G. Gonthier [GM11]. Appendix describes how the source code of our projects is organized.

1.1 THE FLYSPECK PROJECT

In 1611, Johannes Kepler asserted that the maximum density of sphere packing in a three dimensional space is achieved by the familiar cannonball arrangement. This results (known as the Kepler conjecture) can be formulated as follows [Hal12a].

Theorem. *No packing of congruent balls in Euclidean three space has density greater than that of the face-centered cubic packing. This density is* $\pi/\sqrt{18} \approx 0.74$*.*

The conjecture was resolved in 1998 by Thomas Hales and Samuel Ferguson [HF06] (that's why we use the word "theorem" to formulate the conjecture). An important part of the proof of the Kepler conjecture is computer code which cannot be completely verified by a standard peer review process. To eliminate all uncertainties about the correctness of the proof, Hales launched the Flyspeck project in the beginning of 2003 [Hal06a, Hal12b]. The aim of this project is a complete formal verification of the Kepler conjecture. The name of the project is derived from the acronym FPK (the Formal Proof of the Kepler conjecture).

There are three major computationally extensive verification problems in the Flyspeck project. The first problem is a formal generation of special planar graphs. This work has been done by G. Bauer and T. Nipkow [NBS06] in the Isabelle proof assistant [NPW02, Isa].

The second problem is a formal verification of bounds of some linear programs [Hal10]. Partially, this work has been done by S. Obua [Obu05, Obu08, OT09] in Isabelle. In our work, we present another procedure for verification of bounds of linear programs which is done in HOL Light. Our procedure is faster than Obua's method (5 seconds for a single linear program in our work and 8–67 minutes in his work) and it works with the latest revision of the proof of the Kepler conjecture. A complete formal verification of all Flyspeck linear programs is not possible yet. But it will not take a long time to finish this work.

The last problem is a formal verification of about 1000 Flyspeck nonlinear inequalities. A preliminary work has been done by R. Zumkeller [Zum08]. Our work presents a first attempt to verify Flyspeck nonlinear inequalities in a completely formal way.

1.2 FORMAL PROOFS AND HOL LIGHT

Formal proofs are proofs written in a special computer language which can be verified by a computer program. Formal proofs include all, even the most trivial, proof steps. Computer programs which verify formal proofs are called proof assistants. These programs not only check correctness of formal proofs but also help to develop them. A short overview of some formal proof techniques is given in Section 5.1. A good introduction to formal proofs can be found in [Hal08, Har08, Wie08]. Styles of different proof assistants are compared in [Wie06].

In this work, our primary tool is the HOL Light proof assistant [Har10] written by John Harrison. HOL Light belongs to a family of HOL (Higher Order Logic) proof assistants (HOL4 [Hol], Isabelle/HOL [Isa], and others). It is based on simply typed λ calculus [Loa98] and it is written in OCaml programming language [Cam]. HOL Light has a very small and clean core. Every theorem is verified by this core. The simplicity of the core guarantees that the probability of wrong verification results is extremely small. Moreover, J. Harrison verified correctness of HOL Light logic and its implementation [Har06b].

HOL Light contains three fundamental objects: types, terms and theorems. Each term has a type. A theorem is a sequent $p_1, \ldots, p_k \vdash q$ where $p_1, \ldots, p_k, q$ are boolean terms. The terms $p_1, \ldots, p_k$ are called assumptions, the term q is called the conclusion of the theorem. New theorems can be constructed from existing axioms (there are 3 axioms in HOL Light) and inference rules (there are 10 inference rules in HOL Light).

In the text below, we will rarely use raw HOL Light expressions (terms or theorems). All raw HOL Light expressions will be written with a monospaced font. For example, `2 + 2` is a HOL Light term and `|- 2 + 2 = 4` is a HOL Light theorem. In most cases, we will use the corresponding mathematical notation: $2 + 2$ and $\vdash 2 + 2 = 4$.

HOL Light types are defined with the following rules. Each type is either a type variable (denoted by `:A`, `:B`, etc.) or a type constant with a fixed number of arguments. All types are constructed with type variables and three primitive type constants: `fun` (denoted by `->` or $\rightarrow$) with two arguments, `bool` without arguments, and `ind` without arguments. For example, `:A->bool` is a type of functions from an arbitrary type to boolean values.

There are four kinds of terms in HOL Light: variables (e.g., x, y), constants (e.g., 0, sin), applications (if `t` is a term of type `:A->B` and `q` is a term of type `:A`, then the application is defined as `t q` and its type is `:B`; applications are denoted by $t(q)$), and abstractions (if `t` is a term of type `:A` and `x` is a variable of type `:B`, then the abstraction is defined as `\x. t` and its type is `:B->A`; it is denoted by $\lambda x.\ t$). There are two primitive HOL Light constants: the equality `=` of the type `(=):A->A->bool` (the function type `->` is right associative, i.e., `A->A->bool` is the same as `A->(A->bool)`) and Hilbert choice operator `@` (denoted by ε).

All other terms are constructed with these two primitive constants, variables, applications and abstractions. (New constants may be defined as explained below.)

Each term has a type. Terms can be constructed only with primitive core functions which always check if the input term types are compatible. Hence, all HOL Light terms are well-typed by construction.

HOL Light has 10 primitive inference rules which are listed below.

1. Equality is reflexive

$$\texttt{REFL} : \mathit{term} \to \mathit{thm}$$

$$\frac{p}{\vdash p = p}$$

(`REFL ‘x + 2‘`) yields $\vdash (x+2) = (x+2)$.

2. Equality is transitive

$$\texttt{TRANS} : \mathit{thm} \to \mathit{thm} \to \mathit{thm}$$

$$\frac{\Gamma \vdash p = q;\ \ \Delta \vdash q' = r}{\Gamma \cup \Delta \vdash p = r}$$

(here q equals to q' after renaming of bound variables).

(`TRANS ‘|- 2 + 2 = 4‘ ‘|- 4 = 2 * 2‘`) yields $\vdash 2 + 2 = 2 \times 2$.

3. Equal functions applied to equal arguments are equal

$$\texttt{MK\_COMB} : \mathit{thm} \to \mathit{thm} \to \mathit{thm}$$

$$\frac{\Gamma \vdash f = g;\ \ \Delta \vdash x = y}{\Gamma \cup \Delta \vdash f(x) = g(y)}$$

(`MK_COMB(‘(\n. SUC n) = (\n. n + 1)‘,‘2 + 2 = 4‘)`) yields $\vdash \big(\lambda n.\ \mathrm{SUC}(n)\big)(2 + 2) = (\lambda n.\ n + 1)(4)$.

4. The rule of abstraction holds

$$\texttt{ABS} : \mathit{var} \to \mathit{thm} \to \mathit{thm}$$

$$\frac{x;\ \ \Gamma \vdash p = q}{\Gamma \vdash \lambda x.\ p = \lambda x.\ q}$$

(fails if x is free in Γ.)

(`ABS ‘x‘ (REFL ‘x‘)`) yields $\vdash (\lambda x.\ x) = (\lambda x.\ x)$.

5. The application of the function $\lambda x.\ \alpha$ to x gives α

$$\texttt{BETA} : \mathit{term} \to \mathit{thm}$$

$$\frac{(\lambda x.\ \alpha)x}{\vdash (\lambda x.\ \alpha)x = \alpha}$$

(`BETA ‘\n. n + 1) n‘`) yields $\vdash (\lambda n.\ n+1)(n) = n+1$.

6. Assume p, then conclude p

$$\texttt{ASSUME} : \mathit{term} \to \mathit{thm}$$

$$\frac{p}{p \vdash p}$$

(p should be boolean.)

(`ASSUME ‘1 = 0‘`) yields $1 = 0 \vdash 1 = 0$.

7. An equality version of modus ponens

$$\texttt{EQ\_MP} : \mathit{thm} \to \mathit{thm} \to \mathit{thm}$$

$$\frac{\Gamma \vdash p;\ \ \Delta \vdash p' = q}{\Gamma \cup \Delta \vdash q}$$

(`EQ_MP ‘|- T <=> (0 < 1)‘ ‘|- T‘`) yields $\vdash 0 < 1$.

8. Deduces logical equivalence from deduction in both directions

$$\texttt{DEDUCT\_ANTISYM\_RULE} : \mathit{thm} \to \mathit{thm} \to \mathit{thm}$$

$$\frac{\Gamma \vdash p;\ \ \Delta \vdash q}{(\Gamma \setminus q) \cup (\Delta \setminus p) \vdash p = q}$$

(`DEDUCT_ANTISYM_RULE ‘Q |- P‘ ‘P |- Q‘`) yields $\vdash P \iff Q$.

9. Instantiates free variables in a theorem

$$\texttt{INST} : \mathit{list\ of\ pairs\ of\ terms} \to \mathit{thm} \to \mathit{thm}$$

$$\frac{\Gamma \vdash p}{\Gamma[t/x] \vdash p[t/x]}$$

(`INST [‘1‘,‘m:num‘] ‘|- m + n = n + m‘`) yields $\vdash 1 + n = n + 1$.

10. Instantiates types in a theorem

$$\texttt{INST\_TYPE} : \mathit{list\ of\ pairs\ of\ types} \to \mathit{thm} \to \mathit{thm}$$

$$\frac{\Gamma \vdash p}{\Gamma[ty/tv] \vdash p[ty/tx]}$$

There are 3 axioms in HOL Light.

1. Axiom of Extensionality `ETA_AX`

$$\vdash \forall f,\ \big(\lambda x.\ f(x)\big) = f.$$

2. Axiom of Choice `SELECT_AX`

$$\vdash \forall P\, x,\ P(x) \implies P(\varepsilon P).$$

3. Axiom of Infinity `INFINITY_AX`

$$\vdash \exists (f : \mathit{ind} \to \mathit{ind}),\ \text{ONE\_ONE}(f)\ \wedge\ \neg\text{ONTO}(f),$$

where

$$\vdash \text{ONE\_ONE}(f) \iff \big(\forall x\, y,\ f(x) = f(y) \implies x = y\big),$$
$$\vdash \text{ONTO}(f) \iff \big(\forall y,\ \exists x,\ y = f(x)\big).$$

There are two special primitive functions which extend HOL Light with new constants and types. New basic constants can be introduced with the function `new_basic_definition` which accepts one argument: a term in the form `c = t` where `c` must be a variable whose name has not been used as a constant. This function returns a new theorem `|- c = t` where `c` is a new constant.

The function `new_basic_type_definition` defines a new type as a non-empty subset of an existing type.

There are higher-level functions which help to introduce recursively defined functions and types. These functions automatically prove that there exist corresponding basic objects and use primitive extension rules [Har95].

1.3 FORMAL COMPUTATIONS IN HOL LIGHT

What does it mean to compute $2 + 2$ formally? Naturally, we would like to get 4 as the answer. Most informal procedures and programs (for instance, computer algebra systems) are satisfied with this answer. On the other hand, formal procedures must know how the input data and the final result are related. Hence, a formal computation procedure of $2+2$ should return a theorem $\vdash 2 + 2 = 4$ which clearly shows that 4 is the same as $2 + 2$. It is possible to prove the theorem $\vdash 2+2 = 4$ manually but formal computation procedures must be automatic and they must work for different input arguments. There is one more issue. Internally, numerals 2 and 4 are represented in a binary form in HOL Light. Therefore, the theorem $\vdash 2 + 2 = 4$ is actually $\vdash 10_2 + 10_2 = 100_2$. The standard HOL Light procedure `NUM_REDUCE_CONV` simplifies expressions involving binary natural numerals. If we modify the internal representation of natural numerals, then it will be necessary to create new formal computation procedures which will be able to work with new definitions.

A more difficult example is to compute $1 + \pi$ formally. Computer algebra systems either will not do anything with this input or return a decimal approximation of the result. A formal computation procedure should clearly specify what kind of input it expects and what is the form of the result. For instance, the standard HOL Light procedure `REAL_POLY_CONV` will not change the input $1 + \pi$. On the other hand, this procedure simplifies $\pi + \pi$ and returns $\vdash \pi + \pi = 2\pi$. Here, simplification does not mean that the result is simpler than the initial expression. `REAL_POLY_CONV` treats all elements which are not rational constants as indeterminates and converts a given expression into a canonical polynomial form. Suppose we want to get a decimal approximation of $1 + \pi$. Clearly, $1 + \pi = 4.14159$ is not a theorem. On the other hand, it is possible to prove that $\vdash 4.14159 \leq 1+\pi \leq 4.1416$. The result of this formal computation is a theorem containing a pair of rational numbers which approximate $1 + \pi$ below and above.

Based on the examples above, by formal computations we will understand automatic proof procedures. Moreover, results and input arguments of these procedures should be in some well-defined standard forms.

Detailed examples of formal computations in HOL Light can be found in Section 2.1.

1.4 MAIN RESULTS

The main results of this work are formal procedures for proving bounds of linear programs and for verifying multivariate nonlinear inequalities in HOL Light. These formal procedures are based on an efficient implementation of formal arithmetic which is also presented in this work. Another important result is our implementation of SSReflect language (originally developed by G. Gonthier in Coq [GMT11]) in HOL Light. All our results have immediate applications in the Flyspeck project (a formal proof of the Kepler conjecture) [Hal12b, Hal12a].

Our formal verification procedure of bounds of linear programs (Chapter 3) can verify sufficiently large linear programs (more than 1000 constraints and variables) in few seconds. Our method works for general linear programs and for special Flyspeck linear programs. Partial verification of Flyspeck linear programs has been already done by S. Obua [Obu05, Obu08, OT09] in Isabelle. Our verification method is implemented in HOL Light and it is faster than Obua's method (5 second for a single Flyspeck linear programs in our work and 8–67 minutes in his work). Our procedure is not yet capable to verify all Flyspeck linear programs completely. But it will not take a long time to finish this work.

Our formal verification procedure of multivariate nonlinear inequalities is based on interval arithmetic with Taylor approximations (Chapter 4). It works for both polynomial and non-polynomial inequalities (which may contain square roots, arctangents, and arccosines) on rectangular domains. We have successfully tested our formal verification procedure on several simple Flyspeck nonlinear inequalities (we have verified 130 out of 985 inequalities). In theory, almost all Flyspeck inequalities can be verified with our formal verification procedure. Unfortunately, this verification is still too slow: a rough estimate shows that the current formal procedure is about 4000 times slower than the corresponding informal verification algorithm written by Thomas Hales in C++ [Hal03]. With this estimate, it will take more than 4 years to verify all Flyspeck nonlinear inequalities formally on a single computer (the informal procedure requires about 9 hours).

Efficient formal verification procedures require an efficient implementation of formal arithmetic computations (Chapter 2). We present our implementation of formal natural number arithmetic which works with numerals in an arbitrary fixed base. Our implementation improves the performance of arithmetic operations with natural numbers by the factor $\log_2 b$ (where b is a fixed base constant) for linear operations (in the size of input arguments) and by the factor $(\log_2 b)^2$ for quadratic operations. We also describe formal floating-point operations for efficient computations with real numbers. All formal verification results are based on our implementation of interval arithmetic which works with our floating-point numbers.

Our implementation of SSReflect language in HOL Light (SSReflect/HOL Light) provides new opportunities for writing formal proofs in HOL Light with a simple and expressive language (Chapter 5). In particular, SSReflect/HOL Light is the primary formalization tool for all theoretical results in our verification method of nonlinear inequalities. Two important Flyspeck theorems have been proved with SSReflect/HOL Light as well. Another achievement is our complete formalization of Sylow theorems in HOL Light with SSReflect/HOL Light. It is the most advanced abstract algebraic result in HOL Light.

2.0 FORMAL ARITHMETIC

This chapter describes our implementation of formal computations involving numbers. Most of our work described in Chapters 3 and 4 relies on formal numerical computations. An efficient implementation of such computations is essential for successful verification of linear and nonlinear inequalities. Our main goal is to get formal procedures for working with real numbers. In order to achieve this goal, we use a simple and flexible technique of interval arithmetic [Kea96]. Basically, we approximate real quantities by intervals containing these quantities. Our formalization defines intervals of real numbers and operations with them. In our work, all real numbers are approximated by formal floating-point numbers.

Natural number arithmetic is the basis for all other formal numerical computations. So it is not surprising that there are special functions in HOL Light which can simplify basic arithmetic expression of the form $n\ op\ m$ where op is a natural number arithmetic operation. These functions are rather efficient for simple and not extensive computations. Formal arithmetic computations with natural numbers are crucial for serious formal verification procedures. Hence, even more efficient implementation of natural numbers is required. The main idea is to represent natural numbers using a fixed base which is sufficiently large. The standard representation of natural numbers in HOL Light is binary, that is, the base of the representation is 2. It is expected to improve performance of formal natural number arithmetic by the factor of $\log_2(b)$ for linear (in the size of arguments) operations and by the factor of $\log_2(b)^2$ for quadratic operations.

Arbitrary base arithmetic has been already implemented in other proof assistants [Jul08, GT06]. In this work, we go beyond a mere implementation and systematically apply the developed arithmetic for verification of many important results. The work [AGST10] describes a way to perform numerical computations in the Coq proof assistant with native machine arithmetic support. This approach significantly increases performance of formal computations. On the other hand, implementation of these machine-arithmetic assisted operations is only possible by introducing special features in the Coq kernel. One of the most attractive features of HOL Light is its simple kernel. Introduction of new primitive rules is not the best choice for doing formal computations in HOL Light.

The chapter is organized as follows. Section 2.1 describes our implementation of arbitrary base natural number arithmetic in HOL Light. Section 2.2 defines formal floating-point numbers which approximates real numbers and operations with them. Section 2.3 briefly introduces methods for working with integer and rational numbers. Section 2.4 presents our formalization of interval arithmetic in HOL Light. Finally, Section 2.5 contains performance tests of our implementation of formal numerical computations.

2.1 NATURAL NUMBERS

2.1.1 Natural numbers in HOL Light

Natural numbers in HOL Light are defined with three basic objects: the type of natural numbers `:num` and two constants `_0:num` and `SUC:num->num`. The role of these definitions is clear: `_0` is zero (and we will denote this constant by 0) and `SUC` is the successor function (we will denote it by S).

Basic arithmetic operations on natural numbers are defined by recursion. For instance, $0 + n = 0$ for any natural number n and $S(m) + n = S(m + n)$. All other operations are defined in the same way. Addition $(+)$ and multiplication $(\times)$ of natural numbers have all usual properties. Subtraction $(-)$ is a cut-off subtraction, that is, $m - n = 0$ if $m \leq n$. Therefore, some properties of subtraction require additional assumptions, e.g., $\forall m\ n,\ m \leq n \implies (n - m) + m = n$. There is also integer division of natural numbers (DIV) which has the following property

$$\forall n\ m,\ n \neq 0 \implies m = (m \,\mathrm{DIV}\, n) \times n + r \ \wedge\ r < n.$$

Any natural numeral can be represented as a combination of 0 and S. For example, $1 = S(0)$, $2 = S(S(0))$, etc. This representation is canonical. On the other hand, it is very inefficient. Indeed, for a relatively small number 1000 we need at least 1001 constants to represent it. Moreover, formal computations with numerals represented in this way are very slow. As an example, consider formal evaluation of $2 + 1 = S(S(0)) + S(0)$. We have two basic theorems

$$\begin{aligned} \text{addS} &= \vdash m + S(n) = S(m + n), \\ \text{add0} &= \vdash m + 0 = m. \end{aligned}$$

A formal computation of $2 + 1$ can be done with the following inference rules

$r_1 :=$ `INST`$[S(S(0)),\ m;\ 0,\ n]$ addS	$\vdash S(S(0)) + S(0) = S(S(S(0)) + 0)$
$r_2 :=$ `INST`$[S(S(0)),\ m]$ add0	$\vdash S(S(0)) + 0 = S(S(0))$
$t_1 :=$ `AP_TERM` $S\ r_2$	$\vdash S(S(S(0)) + 0) = S(S(S(0)))$
$r_3 :=$ `TRANS` $r_1\ t_1$	$\vdash S(S(0)) + S(0) = S(S(S(0)))$

The left column shows rules with arguments, the right column contains results of each inference rule. `INST` and `TRANS` are primitive HOL Light inference rules (see Section 1.3). `AP_TERM` is a simple rule which is implemented with two primitive inference rules (`REFL` and `MK_COMB`). HOL Light reference manual explains all these rules in details [Har]. In total, we have 5 primitive inference rules to evaluate $2 + 1$ represented as $S(S(0)) + S(0)$. It is easy to generalize this example and construct an algorithm which computes the sum $m + n$ of two terms represented with basic constants S and 0. This algorithm will require $4n + 1$ primitive inference rules (rules for r_1, t_1, and r_3 must be repeated exactly n times; the rule for r_2 is performed once). It is clear, that this algorithm cannot be used for extensive formal computations.

HOL Light provides a more efficient way to represent natural numerals. Each numeral can be represented in a binary form with two special constants `BIT0:num->num` (b_0) and `BIT1:num->num` (b_1). These constants are defined by $b_0(n) = n + n$ and $b_1(n) = S(b_0(n))$. Any natural numeral can be written as a combination of these two constants and 0. For instance, $2 = b_0(b_1(0))$, $3 = b_1(b_1(0))$, $6 = b_0(b_1(b_1(0)))$, etc. This representation corresponds to the usual binary representation of a numeral with the least significant bit first. Note that this representation is not canonical since $0 = b_0(0)$ and hence it is possible to get an arbitrary long representation of 0 and of any other natural numeral. We say that a numeral is in the normal binary form if its representation does not contain subterms of the form $b_0(0)$. Almost all operations which work with binary numerals yield a normalized binary result if the arguments are normalized.

As an example, consider addition of two binary numerals. There are 6 theorems for 4 different cases and 2 terminal cases

$$\begin{aligned}
\text{addb00} &= \vdash b_0(m) + b_0(n) = b_0(m+n),\\
\text{addb01} &= \vdash b_0(m) + b_1(n) = b_1(m+n),\\
\text{addb10} &= \vdash b_1(m) + b_0(n) = b_1(m+n),\\
\text{addb11} &= \vdash b_1(m) + b_1(n) = b_1(S(m+n)),\\
\text{add0} &= \vdash m + 0 = m,\\
\text{add0}' &= \vdash 0 + n = n.
\end{aligned}$$

We also need theorems to compute $S(b_{0,1}(n))$:

$$\begin{aligned}
\text{suc0} &= \vdash S(0) = b_1(0),\\
\text{sucb0} &= \vdash S(b_0(n)) = b_1(n),\\
\text{sucb1} &= \vdash S(b_1(n)) = b_0(S(n)).
\end{aligned}$$

A formal computation of $2 + 1 = b_0(b_1(0)) + b_1(0)$ can be done with the following inference rules

$$\begin{array}{ll}
r_1 := \texttt{INST}[b_1(0),\, m;\, 0,\, n]\ \text{addb01} & \vdash b_0(b_1(0)) + b_1(0) = b_1(b_1(0)+0)\\
r_2 := \texttt{INST}[b_1(0),\, m]\ \text{add0} & \vdash b_1(0) + 0 = b_1(0)\\
t_1 := \texttt{AP\_TERM}\ b_1\ r_2 & \vdash b_1(b_1(0)+0) = b_1(b_1(0))\\
r_3 := \texttt{TRANS}\ r_1\ t_1 & \vdash b_0(b_1(0)) + b_1(0) = b_1(b_1(0))
\end{array}$$

As in the case of addition of $S(S(0)) + S(0)$, we have 5 primitive inference rules. There is no simple formula for the number of primitive inference rules for adding arbitrary binary numerals m and n (the case corresponding to addb11 requires extra inferences to compute $S(a + b)$). However, if we have two binary numerals with at most k binary digits each, then the number of primitive inference rules can be estimated by $Ck + 1$ with $C \leq 10$. For instance, we need at most $10C + 1$ primitive inference rules to compute $1000 + 934$.

2.1.2 Natural numerals with an arbitrary base

The standard binary representation of numerals in HOL Light can be improved by representing each numeral with a fixed base larger than 2. Fix a natural number $b \geq 2$. Then any natural number n can be written as $n = a_0 b^0 + a_1 b^1 + \ldots + a_k b^k$ with $0 \leq a_i < b$ (we don't require $a_k \neq 0$). This representation can be slightly modified and written as $n = a_0 + b(a_1 + b(a_2 + \ldots + b a_k))$. Define b constants in HOL Light `D0, D1, ..., D{b-1}:num->num` (denoted by $d_j(n)$, $j = 0, 1, \ldots, b-1$) as $d_j(n) = j + bn$. Here j and b are fixed numerals (that is, for $b = 10$ we have `|- D2 n = 2 + 10 * n`, `|- D7 n = 7 + 10 * n`, etc.). Any numeral can be written in the form $n = d_{a_0}(d_{a_1} \ldots (d_{a_k}(0)))$ which corresponds to the formula for n given in the beginning of the paragraph.

Note that the base b of the representation must be a fixed constant. It is not possible to change the base after the constants d_j are defined: HOL Light type system does not allow dependent types (that is, types depending on terms) and all definitions are final.

Consider how basic arithmetic operations are implemented for numerals represented with a fixed base b. We will start with the operation which computes the successor $S(n)$ of a given number n. This operation is relatively simple and we will provide all implementation details. For all other arithmetic operations, we will only give the algorithm.

All algorithms will be given in a pseudo-code resembling OCaml programming language [Cam].

Suppose n is a numeral represented by constants `D0`, ..., `D{b-1}` and `0`, then the successor of n can be computed with the following procedure

```
let suc n =
   match n with
   | 0 -> D1 0
   | D{k}(m) when k < b - 1 -> D{k+1}(m)
   | D{b-1}(m) -> D0(suc m)
```

This algorithm is based on the following fact:

$$S(d_k(m)) = (k + bm) + 1 = \begin{cases} (k+1) + bm = d_{k+1}(m), & k + 1 < b, \\ b(n+1) = d_0(S(m)), & k + 1 = b. \end{cases}$$

The formal implementation of the algorithm is the following. First of all, theorems corresponding to the fact above are proved. All theorems have the form $\vdash S(d_k(m)) = d_{k+1}(m)$ for $k < b - 1$ and $\vdash S(d_{b-1}(m)) = d_0(S(m))$. These theorems are saved in a hash table which has constant names as keys and theorems as values (i.e., the key `"D3"` corresponds to the theorem `|- SUC (D3 m) = D4 m`). We have one more theorem for the base case $\vdash S(0) = d_1(0)$. The formal computation algorithm is the following

```
let formal_suc n =
   match n with
   | _0 -> {return |- SUC _0 = D1 _0}
   | D{k}(m) when k < b - 1 ->
       let th0 = {find the theorem in the hash table with the key "D{k}"}
          {return INST[m, m_var] th0}
   | D{b-1}(m) ->
       let th0 = INST[m, m_var] (|- SUC (D{b-1} m) = D0 (SUC m))
       let th1 = formal_suc m
          {return TRANS th0 (AP_TERM D0 th1)}
```

The next example helps to understand this algorithm. Suppose that $b = 10$ and we want to compute the successor of $n = d_9(d_3(d_2(0)))$ ($n = 239$). The algorithm matches n with the last case, i.e., $n = d_9(m)$ (since $9 = 10 - 1$) where $m = d_3(d_2(0))$. The theorem

$\vdash S(d_9(m)) = d_0(S(m))$ is instantiated with $m = d_3(d_2(0))$ and the result is assigned to `th0`. We get

$$\texttt{th0}^1 = \vdash S(d_9(d_3(d_2(0)))) = d_0(S(d_3(d_2(0)))).$$

(The superscript 1 indicates that this variable is defined for the first call of `formal_suc`.) Then, `formal_suc` is called again with new argument $d_3(d_2(0))$. Now, the algorithm matches it with the general case $d_3(m)$ $(3 < 10 - 1)$ and finds the corresponding theorem in the hash table:

$$\texttt{th0}^2 = \vdash S(d_3(m)) = d_4(m).$$

An instance of this theorem is returned where m is replaced with $d_2(0)$. Therefore, we get

$$\texttt{th1}^1 = \vdash S(d_3(d_2(0))) = d_4(d_2(0)).$$

The rule `AP_TERM` converts $\texttt{th1}^1$ into $\vdash d_0(S(d_3(d_2(0)))) = d_0(d_4(d_2(0)))$. Finally, the primitive rule `TRANS` returns

$$\vdash S(d_9(d_3(d_2(0)))) = d_0(d_4(d_2(0))).$$

2.1.3 Addition algorithm

Addition of natural numbers is based on the following facts

$$\begin{aligned} d_i(m) + d_j(n) &= \begin{cases} d_{i+j}(m+n) & i+j < b, \\ d_{i+j-b}(S(m+n)) & i+j \geq b, \end{cases} \\ S(d_i(m) + d_j(n)) &= \begin{cases} d_{i+j+1}(m+n) & i+j+1 < b, \\ d_{i+j+1-b}(S(m+n)) & i+j+1 \geq b. \end{cases} \end{aligned}$$

Theorems corresponding to these facts are generated and saved in hash tables. In order to generate these tables, we start with the result $\vdash d_0(m) + d_0(n) = d_0(m+n)$ which is proved explicitly. Other results are constructed inductively. Suppose we have $\vdash d_i(m) + d_j(n) = d_k(\alpha)$ and $j < b - 1$, then we get $\vdash S(d_i(m) + d_j(n)) = d_i(m) + d_{j+1}(n) = S(d_k(\alpha))$. The explicit value of the expression on the right hand side is taken from the table of theorems for computing the successor S. When we have all results for $0 \leq j \leq b-1$, we increase i and continue until $i = b$. The advantage of this construction is that it is much faster than an explicit proof of each theorem. It is especially important for large values of b (for instance, if $b = 1000$, then it is necessary to construct 10^6 theorems). We construct theorems for computing both $d_i(m) + d_j(n)$ and $S(d_i(m) + d_j(n))$. Later theorems are necessary to avoid successor computations in the addition algorithm. The algorithm is the following

```
let add a b =
   match (a, b) with
   | (0, _) -> a
   | (_, 0) -> b
   | (D{i}(m), D{j}(n)) ->
      if i + j < b then
         D{i + j} (add m n)
      else
         D{i + j - b} (add_c m n)
and
let add_c a b =
   match (a, b) with
   | (0, _) -> formal_suc b
   | (_, 0) -> formal_suc a
   | (D{i}(m), D{j}(n)) ->
      if i + j + 1 < b then
         D{i + j + 1} (add m n)
      else
         D{i + j + 1 - b} (add_c m n)
```

2.1.4 Multiplication algorithm

Multiplication of two natural numbers requires several special tables. First of all, a table of the following results is constructed

$$\vdash d_i(0) \times d_j(0) = d_p(d_q(0)).$$

Values on the right hand side are constructed inductively using the theorem $\vdash m \times S(n) = m \times n + m$, results for formal addition, and the base cases $\vdash m \times d_1(0) = m$.

Based on this table, we construct the next table

$$\vdash d_i(m) \times d_j(n) = d_p(d_q(m \times n) + m \times d_j(0) + n \times d_i(0)).$$

Also, we construct tables for multiplication by $d_j(0)$ on the right and on the left:

$$\begin{aligned} \vdash d_i(m) \times d_j(0) &= d_p(d_q(0) + m \times d_j(0)) \\ \vdash d_j(0) \times d_i(m) &= d_p(d_q(0) + d_j(0) \times d_i(m)) \end{aligned}$$

The multiplication algorithm is the following

```
let mul a b =
   match (a, b) with
   | (0, _) -> 0
   | (_, 0) -> 0
   | (D0(m), _) -> D0 (mul m b)
   | (_, D0(n)) -> D0 (mul a n)
   | (D{i}(0), D{j}(0)) -> D{p}(D{q}(0))
   | (D{i}(0), _) -> mul_left i b
   | (_, D{j}(0)) -> mul_right a j
   | (D{i}(m), D{j}(n)) ->
      let m’ = mul_right m j in
      let n’ = mul_right n i in
         D{p}(D{q}(mul m n) + m’ + n’)
```

Functions `mul_left` and `mul_right` are trivial and they are based on the corresponding multiplication tables.

When we multiply two n-digit numbers with this algorithm, we need to perform approximately n^2 calls of `mul` and `mul_right`. There are more efficient multiplication algorithm for large numbers. For instance, the Karatsuba algorithm [Kar95] requires only $\Theta(n^{\log_2(3)})$ operation to multiply n-digit numbers. We don't need to implement this algorithm because it works well only for large numbers. Our experiments show that for many applications (see the verification of nonlinear inequalities and linear programs) it is only required to work with few-digit numbers (2–4 digits if the base is sufficiently large) for which the Karatsuba algorithm is not the best choice.

2.1.5 Comparison algorithms

Let's define comparison operations for natural numbers. As usual, we generate tables of the following results

$$d_i(m) < d_j(n) \iff \begin{cases} m < n & i \geq j, \\ m \leq n & i < j, \end{cases}$$

$$d_i(m) \leq d_j(n) \iff \begin{cases} m < n & i > j, \\ m \leq n & i \leq j. \end{cases}$$

The comparison algorithm is similar to the addition algorithm and we don't give it here.

It is also important to be able to determine if a given natural number is 0 or not. This is not completely trivial since the representation of natural numbers is not canonical. Nevertheless, the test is simple: $d_i(m) = 0 \iff (i = 0 \ \wedge \ m = 0)$, and the corresponding algorithm is straightforward.

2.1.6 Subtraction and division algorithms

Subtraction of natural numbers has a special implementation. We start from two simple theorems

$$\forall n\ t,\ n + t = m \quad \Longrightarrow \quad m - n = t,$$

$$\forall m\ t,\ m + t = n \quad \Longrightarrow \quad m - n = 0.$$

(Recall that $m - n = 0$ whenever $m \leq n$ for natural numbers in HOL Light.) In order to find $m - n$, it is enough to find t such that $n + t = m$ or $m + t = n$. This number t can be easily found with fast informal machine arithmetic (we use OCaml type `Big_int` for informal computations with natural numbers): it is enough to covert m and n to the corresponding machine numbers $\tilde{m}$ and $\tilde{n}$ and then compute $\tilde{t} = \tilde{m} - \tilde{n}$ (machine arithmetic works with integers). If $\tilde{t} \geq 0$ then construct a formal term for $\tilde{t}$ and call it t, otherwise construct a formal term for $-\tilde{t}$. Then it is left to prove that $n + t = m$ (or $m + t = n$) which can be done with the formal addition procedure described above.

We also define one more procedure for finding the difference of two numbers m and n. This procedure finds $|m - n|$ and returns a theorem $\vdash m \leq n$ or $\vdash n \leq m$. Both results are constructed using an informally computed value t and a formal verification of $n + t = m$ or $m + t = n$. This procedure is useful when one needs to decide which number is bigger before subtracting them.

Integer division of two natural numbers is implemented with the same idea as subtraction. We use the standard HOL Light theorem

$$\forall m\ n\ q\ r,\ (m = qn + r)\ \wedge\ r < n \implies (m\,\mathrm{DIV}\,n = q).$$

To apply this theorem, we need to find q and r which satisfy theorem conditions. They can be quickly found with informal arithmetic. Then it is enough to verify that $qn + r = m$ and $r < n$ with corresponding formal procedures. We could define subtraction in the same way as we defined additions, but it would be quite difficult to implement an efficient formal division algorithm in a direct way. And we don't need such an algorithm: all hard work is done by fast machine arithmetic and we only need to verify that the result is correct. The same idea is used in verification of more difficult problems. We will see how this idea works for finding bounds of linear programs and for proving nonlinear inequalities in next chapters.

2.2 REAL AND FLOATING-POINT NUMBERS

2.2.1 Real numbers and approximations

HOL Light real numbers are formalized directly from natural numbers as a quotient type of nearly additive functions [Har98]. The type of real numbers is called `:real`.

The set of natural numbers is not a subset of real numbers in the sense of the standard subset relation in HOL Light. Nevertheless, there is an injective function `(&):num->real` (denoted by $\mathrm{real}(n)$) which embeds natural numbers into real numbers.

There is no special representation of real numerals in HOL Light. Real numeral can be represented with operations involving natural numerals. For example $\frac{1}{2}$ is represented as `&1 / &2`. There is also a special notation `#0.5` which is equivalent to `&5 / &10`.

Usually, when one needs to work extensively with real numerals, the first step is to define a subset of real numbers which is finite (for real application, like computer programs) or countable (for theoretical constructions). The simplest set may be defined as

$$F = \{n/10^k \mid n \in \mathbb{Z}\}.$$

Here, $k \in \mathbb{N}$ is a fixed number. This set F contains all rational numbers with the fixed denominator. We can think about this set as a set of fixed point approximations (up to k decimal digits after the decimal point) of real numbers. It is not difficult to see that for any real number x there is $y \in F$ such that $|x - y| \le 5 \times 10^{-(k+1)}$. (In computer applications, we consider binary fixed point approximations and we have $n \in [-2^r, 2^r - 1]$.) The main problem of fixed point numbers is that it is easy to lose precision of results of many simple operations with fixed point numbers. For instance, suppose we work with fixed point numbers with 2 decimal digits after the decimal point. Compute an upper bound of $(3.5/1000) \times 100$. The best upper bound of $3.5/1000$ is 0.01. Multiplying it by 100, we get 1 which is pretty far from the exact value 0.35.

A better solution in many cases is a floating-point approximation of real numbers. One example is the IEEE 754 standard [IEE85, Gol90] where real numbers are represented as

$$\pm 2^e m, \quad e \in \mathbb{Z}, m \in \mathbb{N}$$

with certain bounds on the exponent e and the mantissa m.

A completely different way to work with real numbers is to use special Cauchy sequences or streams of digits to represent them [O'C08, O'C09, GNSW07]. The main property of these representations is that they can approximate a given real number as precisely as required.

We are not going to use these constructions since our applications do not require very precise approximations of real numbers.

In examples above, we considered how to approximate real numbers with a fixed subset of real numbers. Now, we need to decide how to perform computations with approximated real numbers. One way is to define operations with approximations and then prove how these operations are related to usual real number operations. For instance, we could define

$$a \oplus b = \inf\{x \mid x \in F \ \wedge \ a + b \leq x\}.$$

It is not difficult to show that $a \oplus b \in F$ and this operations gives the best upper approximation of $a + b$. But this definition tells nothing about actual computation of $a \oplus b$. It would be necessary to give an alternative constructive definition of this operation based on the structure of the set F and then prove that the original definition coincides with the constructive definition. It is not a very simple work [Har99, Har06a]. Moreover, suppose we decide to change the set of approximated values, then all operations and related theorems will also require modifications. Even more, if we don't need a very good approximation of the result (for example, we want to compute faster with a smaller precision), then it will be necessary to define a family of operations depending on some parameter p (precision).

A simpler way to implement formal operations with approximated real numbers is to consider operations which do not yield any formal information about the precision of the result. It is enough to compute an upper and a lower bounds only. We don't need to define any special operations. Instead, for given approximated numbers, we compute a new approximation and return an inequality theorem which tells if the computed approximation is an upper or a lower bound. Now, we are free to use informal procedures to compute approximations. We only need to be able to prove that the results are the desired bounds. In order to get the precision of the computed result, it is enough to consider both upper and lower bounds (i.e., interval arithmetic, see Section 2.4). Also, our informal procedures include a special (informal) parameter p which tells how good the computed approximation should be. We don't claim that the computed approximation is indeed as good as we want it to be. Everything depends on the implementation details and/or bugs in the procedure which computes the approximation. Of course, all procedures are written in a way to give the closest approximations (with one exception for division of two floating-point numbers, see below) but these procedures do not return any formal claims about the precision of results: they just return inequality theorems in the form $\vdash a \odot b \leq r$ or $\vdash r \leq a \odot b$ where $\odot$ is some operation.

2.2.2 Exponential representation of natural numbers

Before defining our representation of floating-point numbers, consider a special exponential representation of natural numbers. We define the constant `num_exp:num->num->num` (denoted by $N_{exp}(m, e)$) as

$$N_{exp}(m, e) = m \times b^e$$

where b is the base of the representation of natural numbers. We call m the mantissa and e the exponent of the exponential representation. The main purpose of this representation is to get exponential approximations of natural numbers with a fixed number of significant digits in the mantissa.

We define a normalization operation which corresponds to the following mathematical operation

$$N_{exp}(m \times b^k, e) = (m \times b^k) \times b^e = m \times (b^{k+e}) = N_{exp}(m, k + e).$$

Here we suppose that m is not divisible by b, unless $m = 0$. In other words, we remove all possible zero digits from the mantissa and increase the exponent.

```
let normalize n =
   match n with
   | num_exp (D0 m) e ->
       let r = normalize m in
       match r with
       | num_exp n e' -> num_exp n (e' + (e + 1))
   | _ -> n
```

This operation returns a canonical exponential representation for any exponential representation of a natural number with one exception for 0. Zero can be represented with an arbitrary exponent. In most cases, this is not a problem since formal natural number operations always return canonical results, so it should almost never happen that `0 = D{k} 0` and the exponential representation will also be canonical. A denormalized 0 could appear when a natural number is approximated below by another natural number. The denormalization is not a big issue unless the exponent of a denormalized zero starts to increase. The main source of the growth of the exponent of a denormalized zero is multiplication of a denormalized zero by another number. This is prevented by considering a special case for multiplication of floating-point numbers.

In the current implementation of formal floating-point operations, it is also necessary to be able to denormalize the exponential representation of a natural number, that is, write

$$N_{exp}(m, e) = m \times b^e = (m \times b^e) \times b^0 = N_{exp}(m \times b^e, 0).$$

This operation is simple and the denormalization algorithm is obvious.

The main operation for natural numbers in the exponential form is the approximation with a fixed number of digits in the mantissa. The lower approximation is computed by the algorithm

```
let lo_num_conv p n =
   let k = {number of digits in n}
      if k <= p then n
      else
         let lo_bound k n =
            match n with
            | D{k} m ->
               if k > 1 then
                  D0 (lo_bound (k - 1) m)
               else
                  D0 m
         lo_bound (k - p) n
```

The upper approximation is similar. However, we have two algorithms for computing non-strict and strict upper bounds (the latter is necessary in our implementation of the division algorithm).

We define arithmetic operations with natural numbers in the exponent form which return exact results. These results are approximated below and above with the approximation procedures and lower and upper bounds with at most p digits in the mantissa are returned.

Operations with exponential natural numbers are based on the following theorems

$$\begin{aligned}
n_1 b^{e_1} \times n_2 b^{e_2} \quad &= \quad (n_1 \times n_2) b^{e_1+e_2}, \\
n_1 b^{e_1} + n_2 b^{e_2} \quad &= \quad \begin{cases} (n_1 + n_2 b^{e_2-e_1}) b^{e_1} & e_1 \leq e_2, \\ (n_1 b^{e_1-e_2} + n_2) b^{e_2} & e_2 \leq e_1, \end{cases} \\
n_1 b^{e_1} - n_2 b^{e_2} \quad &= \quad \begin{cases} (n_1 - n_2 b^{e_2-e_1}) b^{e_1} & e_1 \leq e_2, \\ (n_1 b^{e_1-e_2} - n_2) b^{e_2} & e_2 \leq e_1, \end{cases} \\
n_1 b^{e_1} \leq n_2 b^{e_2} \quad &\iff \quad \begin{cases} n_2 e^{e_2-e_1} \leq n_1 & e_1 \leq e_2, \\ n_2 \leq n_1 b^{e_1-e_2} & e_2 \leq e_1, \end{cases} \\
n_1 b^{e_1} \text{ DIV } n_2 b^{e_2} \quad &= \quad \begin{cases} n_1 \text{ DIV } n_2 b^{e_2-e_1} & e_1 \leq e_2, \\ n_1 b^{e_1-e_2} \text{ DIV } n_2 & e_2 \leq e_1. \end{cases}
\end{aligned}$$

The corresponding algorithms are straightforward. Results on the right hand side are computed with formal natural number arithmetic. Numbers in the form nb^e on the right hand side are denormalized first (e.g., `num_exp (D3 0) (D2 0)` is converted to `D0 (D0 (D3 0))`). Note that denormalization is necessary only for numbers in the form $b^{e_1-e_2}$ ($b^{e_2-e_1}$). In most real applications, e_1 and e_2 are not very different and denormalization will add only a few more digits. Nevertheless, a better implementation is possible which we will briefly discuss in Section 2.5.

2.2.3 Formal floating-point numbers

We introduce a new constant `float_num:bool->num->num->real` (denoted by $\mathrm{float}(s, m, e)$) which defines a real number with the following formula

$$\mathrm{float}(s,m,e) = \begin{cases} \mathrm{real}\big(N_{exp}(m,e)\big)/\mathrm{real}\big(N_{exp}(b,e_0)\big) & \text{if } s \text{ is false,} \\ -\mathrm{real}\big(N_{exp}(m,e)\big)/\mathrm{real}\big(N_{exp}(b,e_0)\big) & \text{if } s \text{ is true.} \end{cases}$$

The corresponding HOL Light definition is

```
|- float_num s n e =
     (if s then (-- &1) else (&1)) * &(num_exp n e) / &(num_exp b min_exp)
```

The constant `min_exp` (denoted by e_0) is a fixed natural number constant which specifies the minimal exponent for floating-point numbers. Mathematically, the HOL Light term `float_num F n e` represents nb^{e-e_0} and `float_num T n e` represents $-nb^{e-e_0}$.

All formal operations with floating-point numbers are implemented as procedures which take terms containing floating-point numbers as arguments and return inequality theorems which give upper or lower bounds of results. Also, these procedures accept the precision parameter which restricts the number of significant digits in the results. For example, the following procedure `float_add_hi 2 a b` returns $\vdash a+b \le c$ for some floating-point c which contains at most 2 significant digits in the mantissa.

Implementations of most formal floating-point operations are straightforward. Essentially, it is necessary to do case splitting on the signs of arguments, compute the corresponding result with exponentially represented natural numbers, and normalize the exponent of the result.

Suppose we have to add two floating-point numbers $f_1 = \pm n_1 b^{e_1 - e_0}$ and $f_2 = \pm n_2 b^{e_2 - e_0}$. We use the following facts to find a lower bound of the sum

$$f_1 + f_2 \ge \begin{cases} nb^{e-e_0} & nb^e \le n_1 b^{e_1} + n_2 b^{e_2} \ \wedge\ 0 \le f_1 \ \wedge\ 0 \le f_2, \\ -nb^{e-e_0} & n_1 b^{e_1} + n_2 b^{e_2} \le nb^e \ \wedge\ f_1 \le 0 \ \wedge\ f_2 \le 0, \\ nb^{e-e_0} & nb^e \le n_1 b^{e_1} - n_2 b^{e_2} \ \wedge\ n_2 b^{e_2} \le n_1 b^{e_1} \ \wedge\ 0 \le f_1 \ \wedge\ f_2 \le 0, \\ -nb^{e-e_0} & n_2 b^{e_2} - n_1 b^{e_1} \le nb^e \ \wedge\ n_1 b^{e_1} \le n_2 b^{e_2} \ \wedge\ 0 \le f_1 \ \wedge\ f_2 \le 0. \end{cases}$$

The case $f_1 \leq 0$ and $0 \leq f_2$ is eliminated by commutativity: $f_1 + f_2 = f_2 + f_1$. The upper bound approximations are similar. The implementation of the addition algorithm for floating-point numbers is straightforward. This algorithm selects an appropriate theorem based on the signs of arguments and then performs the corresponding operation with exponentially represented natural numbers.

Subtraction of floating-point numbers is implemented via addition: $f_1 - f_2 = f_1 + (-f_2)$.

Multiplication of floating-point numbers $f_1 = \pm n_1 b^{e_1 - e_0}$ and $f_2 = \pm n_2 b^{e_2 - e_0}$ is almost the same as addition. We have

$$f_1 \times f_2 = \pm \frac{n_1 b^{e_1} \times n_2 b^{e_2}}{b^{e_0 + e_0}} = \pm n b^{(e - e_0) - e_0}.$$

Here, $nb^e = n_1 b^{e_1} \times n_2 b^{e_2}$. As we see, to get a lower (upper) bound of the product of f_1 and f_2, it is enough to compute a bound of $n_1 b^{e_1} \times n_2 b^{e_2}$ (which bound to compute depends on the signs of f_1 and f_2) and then subtract e_0 from the exponent of this bound. The implementation of the algorithm is obvious. Note that the current implementation of the algorithm fails if $e < e_0$. This situation may happen when two small (close to zero) floating-point numbers are multiplied together. We have never seen this situation in our real tests and applications. Nevertheless, future implementations of the formal floating-point arithmetic library will address this issue.

Consider the division algorithm for positive floating-point numbers $f_1 = n_1 b^{e_1 - e_0}$ and $f_2 = n_2 b^{e_2 - e_0}$, $n_2 \neq 0$. We have

$$\frac{f_1}{f_2} = \frac{n_1 b^k}{n_2} b^{(e_1 + e_0 - e_2 - k) - e_0}.$$

Here, k is any natural number. Compute q with formal integer division such that $n_1 b^k = q n_2 + r$ with $r < n_2$. We get

$$\frac{n_1}{n_2} - \frac{q}{b^k} = \frac{n_1 b^k - q n_2}{n_2 b^k} = \frac{r}{n_2 b^k} < \frac{1}{b^k}.$$

In our algorithm, we set $k = p$ where p is the precision parameter. The bounds of the result are obtained by approximating q below and above with at most p digits: $n_l b^{e_l} \leq q < n_u b^{e_u}$ with $n_l, n_u < b^p$. Note that the upper bound must be strict since $q \leq nb^e$ does not imply an upper bound inequality for f_1/f_2.

The division algorithm computes q, takes its lower or upper bound in the form nb^e and then the final result is $nb^{(e_1+e_0+e-e_2-k)-e_0}$. If $e_1+e_0+e < e_2+k$, then the algorithm fails in the same way as the multiplication algorithm. The division algorithm also fails if $n_2 = 0$.

The last operation which we define for floating-point numbers is the square root operation. The square root algorithm based on the following theorem

$$(n_1 b^{e_1})^2 \le nb^{2k} \le (n_2 b^{e_2})^2 \implies n_1 b^{(e_1+e/2+e_0/2-k)-e_0} \le \sqrt{nb^{e-e_0}} \le n_2 b^{(e_2+e/2+e_0/2-k)-e_0}.$$

For a given non-negative floating-point number $f = nb^e$, the algorithm computes the integer square root of nb^{2p} (where p is the precision parameter) with informal machine arithmetic. Then this number is approximated below and above with at most p digits such that the conditions of the theorem are satisfied. And then the final result is computed. Note that the theorem requires the exponent e of the argument to be even (since we need to be able to compute $e/2$ exactly). When e is odd, then the argument is transformed by $nb^e = (nb)b^{e-1}$ and then the square root is taken.

2.2.4 Cached arithmetic

There is one more optimization trick which increases the performance of formal floating-point and natural arithmetic. We store results of all basic arithmetic operations with natural and floating-point numbers in a cache. Each operation has its own cache and each cache is a hash table where keys are strings which encodes arguments of an operation. The table contains results of the operation, i.e., equality and inequality theorems. Each hash table has limited size and when this size is exceeded, then the entire table is cleared. This naive approach may be replaced later with a more sophisticated technique which removes some entries of the table only. The size of the cache can be changed with a parameter. Also, caching can be turned off but it is useful only for performance tests. For a randomly generated test, caching does not increase the performance (but it doesn't decrease it since all caching operations are fast compared with formal computations). Real tests show much better results: a formal verification of a nonlinear inequality with cached arithmetic could be two times faster than the same verification without caching.

2.3 INTEGER AND RATIONAL NUMBERS

There is a special type of integers in HOL Light. But it is more convenient to work with integers as a subset of real numbers. Every integer number is represented by a natural number corresponding to its absolute value and by its sign. For instance, -2 can be represented as `--(&2)` in HOL Light where `--` is the negation of real numbers in HOL Light. Formal computations with integers are simple. Almost all work is done by formal natural number arithmetic. It is only necessary to deal with different cases depending on the signs of arguments.

Addition of integers is based on the following obvious theorems for natural numbers m and n:

$$\mathrm{real}(m) + \mathrm{real}(n) = \mathrm{real}(m+n), \tag{2.1}$$

$$-\mathrm{real}(m) + -\mathrm{real}(n) = -\mathrm{real}(m+n), \tag{2.2}$$

$$-\mathrm{real}(m) + \mathrm{real}(m+n) = \mathrm{real}(n), \tag{2.3}$$

$$-\mathrm{real}(m+n) + \mathrm{real}(m) = -\mathrm{real}(n), \tag{2.4}$$

$$-\mathrm{real}(m+n) + \mathrm{real}(m) = -\mathrm{real}(n), \tag{2.5}$$

$$\mathrm{real}(m+n) + -\mathrm{real}(m) = \mathrm{real}(n). \tag{2.6}$$

If the arguments have the same sign, then the result follows from cases 2.1 and 2.2. It is only necessary to compute $m+n$ formally using natural number arithmetic. If the signs of arguments are different, then the procedure is different. Suppose that arguments are x and y such that $x < 0$ and $0 \le y$. Also, suppose $x = -\mathrm{real}(m)$ and $y = \mathrm{real}(k)$. The first step is to find the corresponding machine natural numbers (informal natural numbers) $\tilde{x}$ and $\tilde{y}$ which correspond to absolute values of x and y, i.e., to m and k. Then, we use fast machine operations to compare these two numbers and find their difference. Suppose $\tilde{x} \le \tilde{y}$, then we find $\tilde{z}$ such that $\tilde{y} = \tilde{x} + \tilde{z}$. Construct a formal natural term for $\tilde{z}$ and call it n. Then we have to prove that $m + n = k$ with formal natural number arithmetic. The final result follows from case 2.3 of the theorem above, i.e., $\vdash x + y = \mathrm{real}(n)$. Other cases are processed in the same way. Subtraction of integers is done via addition by first switching the sign of the second argument.

Multiplication of integers is simpler. Again, we need to consider several cases

$$\begin{aligned}
\mathrm{real}(m) \times \mathrm{real}(n) &= \mathrm{real}(m \times n),\\
-\mathrm{real}(m) \times -\mathrm{real}(n) &= \mathrm{real}(m \times n),\\
-\mathrm{real}(m) \times \mathrm{real}(n) &= -\mathrm{real}(m \times n),\\
\mathrm{real}(m) \times -\mathrm{real}(n) &= -\mathrm{real}(m \times n).
\end{aligned}$$

We only need to compute $m \times n$ formally and do case splitting on the signs of arguments. It is not required to perform any informal arithmetic operations in order to find the final result.

Formal integer operations are only required for our verification procedure of linear programs. Hence, we don't need to define division and other operations.

There is no special type of rational numbers in HOL Light. Any rational number can be represented as `&m / &n` or `--(&m / &n)` in HOL Light with m, n natural numbers such that $n \neq 0$. Formal operations with rational numbers are very similar to integer operations. The main difference is that we want to keep rational numbers in lower terms. Hence, it is necessary to cancel all common factors after performing arithmetic operations. All cancellations can be done informally (which is much faster than a formal implementation). For instance, addition of rational numbers is done with the following theorem

$$\begin{aligned}
&0 < n_1 \ \wedge\ 0 < n_2 \ \wedge\ 0 < n_3\\
&\qquad\qquad \wedge\ \big(x_1 \times \mathrm{real}(n_2) + x_2 \times \mathrm{real}(n_1)\big) \times \mathrm{real}(n_3) = x_3 \times \mathrm{real}(n_1) \times \mathrm{real}(n_2)\\
&\qquad\qquad\qquad\qquad \Longrightarrow\ \frac{x_1}{\mathrm{real}(n_1)} + \frac{x_2}{\mathrm{real}(n_2)} = \frac{x_3}{\mathrm{real}(n_3)}.
\end{aligned}$$

To apply this theorem, it is necessary to compute x_3 and n_3 informally and then prove the hypotheses of the theorem with formal integer arithmetic.

We do not use formal rational arithmetic in verification methods described in next chapters. Instead, we work with floating-point numbers which form a special class of rational numbers. The main problem of rational number arithmetic is that numerators and denominators of rational numbers grow very fast. It leads to slow computations. On the other hand, basic arithmetic operations are exact for rational numbers. It is not a significant advantage over floating-point numbers since results of many important operations (square root, arctangent, etc.) are not rational numbers.

2.4 INTERVAL ARITHMETIC

2.4.1 Interval approximations

Interval arithmetic is a well known technique for performing reliable computations [Kea96]. The idea of interval arithmetic is very simple: for each real number x we can find rational (or floating-point) lower and upper bounds $\underline{x}$ and $\overline{x}$ such that $x \in [\underline{x}, \overline{x}]$. Moreover, the lower and upper bounds can be arbitrary close to x, i.e., $\overline{x} - \underline{x}$ is arbitrary small. The main purpose of interval arithmetic is to prove that some arithmetic expression is bounded by some numbers. To solve this problem, we need to be able to perform arithmetic operations on intervals. Given a function $f : \mathbb{R} \to \mathbb{R}$ and an interval $[\underline{x}, \overline{x}]$, we need to find a and b such that $f(x) \in [a, b]$ for any $x \in [\underline{x}, \overline{x}]$. Usually, $a = \underline{f}(\underline{x}, \overline{x})$ and $b = \overline{f}(\underline{x}, \overline{x})$ for some functions $\overline{f}$ and $\underline{f}$. For a binary operation $g : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, we have the same problem where the final result is $g(x, y) \in [a, b]$ for any $x \in [\underline{x}, \overline{x}]$ and $y \in [\underline{y}, \overline{y}]$. Finally, we need to be able to compare an interval $[a, b]$ with some bounds l, u. This operation is simple: if $e \in [a, b]$, $l \le a$, and $b \le u$ then $l \le e \le u$. Our formalization of floating-point numbers immediately yields interval approximations for some basic operations (addition, subtraction, square root); other operations (multiplication, division) require case splitting on the signs of interval bounds.

Formalization of interval arithmetic starts with the definition of an interval approximation. We need to know not only the end points of an interval but also a value which is approximated by this interval. Hence, our main HOL Light definition is a predicate

```
|- interval_arith x (a,b) <=> a <= x /\ x <= b
```

In other words, `interval_arith x (a,b)` means $x \in [a, b]$ (and we will use this notation).

Whenever we have a theorem $\vdash x \in [a, b]$, this theorem implicitly asserts that $a \le b$, i.e., the interval is not empty.

2.4.2 Basic interval operations

When we work with interval approximations in the form $x \in [a, b]$, we always assume that x could be any HOL Light term of the type `:real`. a, b should be floating point numerals for all computations involving interval approximations. Each formal interval operation takes interval theorems as arguments and returns a new interval theorem which approximates the result of an operation. For example, if `th1` is $\vdash \pi \in [3, 4]$ and `th2` is $\vdash \sqrt{2} \in [1, 2]$ (3, 4, 1, and 2 should be represented with floating-point numbers here), then the operation `float_interval_mul 2 th1 th2` returns $\vdash (\pi \times \sqrt{2}) \in [3, 8]$. Since each interval operation is based on the corresponding floating-point operation, there is the precision parameter which is passed directly to floating-point operations.

Negation of an interval is simple.

$$x \in [a, b] \implies -x \in [-b, -a].$$

Note that the formal negation operation does not take the precision parameter since the corresponding floating-point operation does not have it either.

Addition and subtraction of intervals are also simple.

$$\begin{aligned} x \in [a, b] \ \wedge \ y \in [c, d] \ \wedge \ l \le a \pm c \ \wedge \ b \pm d \le u& \\ \implies x \pm y \in [l, u].& \end{aligned}$$

We use formal floating-point operations to compute lower and upper bounds for end points of the result.

Multiplication requires to consider several cases.

$$x \in [a, b] \ \wedge \ y \in [c, d] \implies x \times y \in I,$$

$$I = \begin{cases} [0,0] & \big(0 \le a \,\land\, b \le 0\big) \,\lor\, \big(0 \le c \,\land\, d \le 0\big), \\ [l,u] & l \le ac \,\land\, bd \le u \,\land\, 0 \le a \,\land\, 0 \le c, \\ [l,u] & l \le bd \,\land\, ac \le u \,\land\, b \le 0 \,\land\, d \le 0, \\ [l,u] & l \le bc \,\land\, ad \le u \,\land\, 0 \le a \,\land\, d \le 0, \\ [l,u] & l \le ad \,\land\, bc \le u \,\land\, b \le 0 \,\land\, 0 \le c, \\ [l,u] & l \le ad \,\land\, bd \le u \,\land\, a \le 0 \le b \,\land\, 0 \le c, \\ [l,u] & l \le bc \,\land\, ac \le u \,\land\, a \le 0 \le b \,\land\, d \le 0, \\ [l,u] & l \le bc \,\land\, bd \le u \,\land\, 0 \le a \,\land\, c \le 0 \le d, \\ [l,u] & l \le ad \,\land\, ac \le u \,\land\, a \le 0 \,\land\, c \le 0 \le d, \\ [\min\{l_1,l_2\}, \max\{u_1,u_2\}] & l_1 \le ad \,\land\, l_2 \le bc \,\land\, ac \le u_1 \,\land\, bd \le u_2 \\ & \land\, a \le 0 \le b \,\land\, c \le 0 \le d. \end{cases}$$

Internally, the case splitting is based on the first parameter of the `float_num` constant. If the first parameter is `T` then the corresponding value is assumed to be non-positive, otherwise it is non-negative. Thus, it is necessary to consider special cases. First two cases correspond to a special situation when $[a,b] = [0,-0]$. Clearly, $-0 = 0$ but it has the representation `float_num T 0 e` and hence $-0 \le 0$. In most cases, it is only necessary to perform two formal floating-point operations for computing new lower and upper bounds.

Interval division is very similar to multiplication:

$$x \in [a,b] \,\land\, y \in [c,d] \implies x/y \in I,$$

$$I = \begin{cases} [0,0] & 0 \le a \,\land\, b \le 0 \\ [l,u] & l \le a/d \,\land\, b/c \le u \,\land\, 0 \le a \,\land\, 0 < c, \\ [l,u] & l \le b/c \,\land\, a/d \le u \,\land\, b \le 0 \,\land\, d < 0, \\ [l,u] & l \le b/d \,\land\, a/c \le u \,\land\, 0 \le a \,\land\, d < 0, \\ [l,u] & l \le a/c \,\land\, b/d \le u \,\land\, b \le 0 \,\land\, 0 < c, \\ [l,u] & l \le a/c \,\land\, b/c \le u \,\land\, a \le 0 \le b \,\land\, 0 < c, \\ [l,u] & l \le b/d \,\land\, a/d \le u \,\land\, a \le 0 \le b \,\land\, d < 0. \end{cases}$$

(Note that the first case does not require any conditions on the interval $[c, d]$ since $0/0 = 0$ in HOL Light.) When $c \leq 0 \leq d$, the procedure fails and reports division by zero.

The square root function is monotone, so the formal interval operation for the square root simply follows from the theorem

$$x \in [a, b] \ \wedge \ 0 \leq a \ \wedge \ l \leq \sqrt{a} \ \wedge \ \sqrt{b} \leq u \implies \sqrt{x} \in [l, u].$$

2.4.3 Arctangent and arccosine

In order to be able to verify nonlinear Flyspeck inequalities, we need procedures for interval approximations of arctangent and arccosine. For both functions, we do not define how to compute upper and lower bounds separately.

The following results were formalized by Thomas Hales in his formal proof of Jordan curve theorem [Hal07].

Define

$$\begin{aligned}
\mathrm{halfatn}(x) &= \frac{x}{\sqrt{1 + x^2} + 1},\\
\mathrm{halfatn}_4(x) &= (\mathrm{halfatn} \circ \mathrm{halfatn} \circ \mathrm{halfatn} \circ \mathrm{halfatn})(x),\\
\mathrm{halfatn}_4^{co}(x, j) &= \frac{(-1)^j \mathrm{halfatn}_4(x)^{(2j+1)}}{2j + 1}.
\end{aligned}$$

Then the following results can be proved

$$\begin{aligned}
&\forall x,\ |\mathrm{halfatn}(x)| < 1,\\
&\forall x\ t,\ |x| < t \implies |\mathrm{halfatn}(x)| < \frac{t}{2},\\
&\forall x,\ \arctan(x) = 2 \arctan(\mathrm{halfatn}(x)),\\
&\forall n\ x,\ \left|\arctan(x) - 16 \sum_{j=0}^{n} \mathrm{halfatn}_4^{co}(x, j)\right| \leq 2^{-(6n+5)}.
\end{aligned}$$

The interval approximation of arctangent is computed with the theorem

$$\begin{aligned}
16 \sum_{j=0}^{n} \mathrm{halfatn}_4^{co}(x, j) \in [a, b] \ \wedge \ 2^{-(6n+5)} \in [c, d]&\\
\wedge\, b + d \leq h \ \wedge \ l \leq a - d \implies \arctan(x) \in [l, h].&
\end{aligned}$$

The interval for the sum in this theorem is computed as follows. We define polynomial expressions

$$\begin{aligned}
&\mathrm{poly}([], x) = 0 \ \wedge\ \mathrm{poly}(h :: t, x) = h + x \times \mathrm{poly}(t, x),\\
&\mathrm{poly}_{even}(c, x) = \mathrm{poly}(c, x \times x),\\
&\mathrm{poly}_{odd}(c, x) = x \times \mathrm{poly}_{even}(c, x).
\end{aligned}$$

(The corresponding HOL Light constants are `poly_f`, `poly_f_even`, and `poly_f_odd`.) The first argument of these functions is a list of polynomial coefficients. The notation $[]$ denotes an empty list, $h :: t$ denotes a list obtained by attaching an element h in front of a list t. These polynomial expressions can be efficiently evaluated if we know intervals for all coefficients c and an interval for a variable x.

We define a list of first $n + 1$ coefficients of the arctangent approximation

$$\mathrm{atn}_{table}(n) = \left[1; -\frac{1}{3}; \ldots; \frac{(-1)^n}{2n+1}\right].$$

Then we get

$$\forall x\ n,\ \sum_{j=0}^{n} \mathrm{halfatn}_4^{co}(x, j) = \mathrm{poly}_{odd}\big(\mathrm{atn}_{table}(n), \mathrm{halfatn}_4(x)\big).$$

We precompute values of n (the number of terms in the sum) for different values of the precision parameter p. For each p, we find n_p such that

$$2^{-(6n_p+5)} \le b^{-(p+1)}.$$

Then we compute interval approximations of $2^{-(6n_p+5)}$ and interval approximations for coefficients $\mathrm{atn}_{table}(n_p)$ (we use $p + 1$ as the precision parameter for these computations).

When we need to compute arctangent for the given interval approximation $x \in [\underline{x}, \overline{x}]$ with precision p, we get the parameter $n = n(p)$ and the approximation of $2^{-(6n+5)}$ from the computed tables. We also get interval approximations of the coefficients. Then we compute the interval for $\mathrm{halfatn}_4(x)$ and all these intervals to evaluate $\mathrm{poly}_{odd}\big(\mathrm{atn}_{table}(n), \mathrm{halfatn}_4(x)\big)$. It is left to multiply the result by 16 and to compute lower and upper bounds of the approximation of arctangent.

Arccosine is computed with the following formula

$$-1 < x < 1 \implies \arccos x = \frac{\pi}{2} - \arctan\left(\frac{x}{1-x^2}\right).$$

The interval bounds are given by the following theorem

$$\frac{\pi}{2} - \arctan\left(\frac{x}{1-x^2}\right) \in I \ \wedge\ 1 - x^2 \in [l, h] \ \wedge\ 0 < l \implies \arccos x \in I.$$

We compute an interval for $1 - x^2$, verify that its lower bound is positive and then compute the interval for the approximation of arccosine.

2.4.4 Interval evaluation of HOL Light expressions

We have a function which converts a given HOL Light term into a special procedure which can efficiently compute interval approximations of the given term. Input terms may contain free variables which become arguments of the evaluation procedure. If the input term $f(x)$ has a free variable x, then the corresponding argument should be a theorem $\vdash x \in [a, b]$ where a and b are floating-point numbers. The evaluation procedure will compute bounds of $f(x)$ and return a theorem $\vdash f(x) \in [f^{lo}, f^{hi}]$. To optimize computations for different values of arguments, it is possible to precompute all constant expressions inside the input term and fix their values for further evaluations. It is required to specify the precision for approximation of constants. For instance, if the input term is $\pi/2 + x$, then it is possible to evaluate and fix bounds of $\pi/2$ before doing any other formal computations with this expression.

Another simple optimization procedure replaces all common subexpressions in a given list of input terms. A special evaluation procedure computes these common subexpressions first and then refers to computed results when the corresponding functions are evaluated. This optimization procedure does not take into account any properties of operations (associativity, commutativity). For instance, if we have two expressions $x \times x + y$ and $x \times x - 3$, then the optimization procedure finds a common subterm $x \times x$. When we evaluate these two terms for given approximations of x and y, the expression $x \times x$ will be evaluated first (call the result t) and then the procedure will compute $t + y$ and $t - 3$.

All interval arithmetic procedures take interval theorems as arguments. If we want to find an interval approximation of a function $f(x)$ on an interval $x \in [a, b]$, then we need first to create an interval theorem for x. It can be done with HOL Light primitive inference rule `ASSUME` which returns a theorem for any term t in the form $t \vdash t$ (we have t as the assumption and as the conclusion of this theorem). Therefore, we can create a theorem $x \in [a, b] \vdash x \in [a, b]$. This theorem is a perfectly fine argument for formal interval evaluation functions. If we apply a procedure for evaluation of f, we get $x \in [a, b] \vdash f(x) \in [c, d]$ where c, d are interval bounds for $f(x)$. Now, we can discharge the assumption and generalize x to get $\vdash \forall x,\ x \in [a, b] \implies f(x) \in [c, d]$.

2.5 RESULTS AND TESTS

2.5.1 Possible improvements

The current implementation of formal floating-point arithmetic is quite efficient. However, there are several things that can be improved. First of all, an alternative definition of floating-point numbers may be considered where the exponent is an integer number. The advantage of integer exponents is that underflow never happens (i.e., the situation when the exponent of the result is less than some fixed number e_0) and some operations are simpler (no normalization of the exponent is required).

Floating-point division and square root algorithms can be improved by selecting the parameter k adaptively based on the size of input arguments. The current implementation always takes $k = p$ which is not an optimal value for some input arguments.

The addition and subtraction algorithms may also be improved. They are not very efficient when the difference between exponents of the arguments is quite big. Right now, the exact sum (difference) is computed first and then it is rounded to a desired precision. Computation of the exact result is excessive when the exponents of arguments differs more than the requested precision.

2.5.2 Performance tests

This section contains performance test results for formal arithmetic operations. All tests were performed on Intel Core i5, 2.67GHz running Ubuntu 9.10 inside Virtual Box 4.2.0 on a Windows 7 host; the Ocaml version was 3.09.3; the caching was turned off. The effect of cached arithmetic is shown in Section 4.7.

Results of performance tests for formal natural number arithmetic operations with an arbitrary base are given in Tables 1, 2, and 3. These results were obtained by performing formal operations on 1000 pairs of randomly generated numbers. For the division operation, the second argument had 5 decimal digits always to get nontrivial results.

Performance test results for floating-point operations are shown in Table 4. The results were obtained by performing formal floating-point operations on randomly generated input arguments. Both lower and upper bounds were computed in all tests. The natural number arithmetic base was $b = 200$ and the precision parameter was $p = 5$ for all these tests.

The next test compares floating-point arithmetic with rational arithmetic by evaluating the polynomial expression

$$f(x) = 1 + x\left(1 + x\left(\frac{1}{2} + x\left(\frac{1}{6} + x\left(\frac{1}{24} + x\left(\frac{1}{120} + \frac{1}{720}x\right)\right)\right)\right)\right)$$

with floating-point and rational operations for different values of x. Test results are given in Table 5.

Table 1: Performance results for 1000 addition operations

Size of operands	Native HOL Light (s)	Base 16 (s)	Base 256 (s)
5 decimal digits	0.157	0.069	0.039
10 decimal digits	0.265	0.076	0.044
15 decimal digits	0.417	0.104	0.064
20 decimal digits	0.529	0.137	0.078
25 decimal digits	0.673	0.167	0.097

Table 2: Performance results for 1000 multiplication operations

Size of operands	Native HOL Light (s)	Base 16 (s)	Base 256 (s)
5 decimal digits	2.180	0.384	0.137
10 decimal digits	6.533	1.331	0.377
15 decimal digits	15.002	3.359	1.163
20 decimal digits	57.239	5.995	2.015
25 decimal digits	82.600	9.303	3.187

Table 3: Performance results for 1000 division operations

Size of operands	Native HOL Light (s)	Base 16 (s)	Base 256 (s)
5 decimal digits	1.612	0.222	0.171
10 decimal digits	4.154	0.643	0.299
15 decimal digits	6.972	1.193	0.525
20 decimal digits	9.735	1.647	0.701
25 decimal digits	13.071	2.157	0.894

Table 4: Performance results for 1000 floating-point operations

Size of the mantissa	Add (s)	Mult (s)	Div (s)	Sqrt (s)	Arctan (s)
5 decimal digits	1.421	0.583	2.076	2.938	70.848
10 decimal digits	1.588	1.491	2.744	3.061	71.439
15 decimal digits	1.926	3.469	3.850	3.216	73.797
20 decimal digits	2.209	5.572	4.601	3.343	77.173
25 decimal digits	2.522	8.215	5.580	3.540	84.163

Table 5: Performance results for 100 evaluations of a polynomial expression

Method	$x = 1$	$x = 1/3$	$x = 234451/1234567$
rational	1.804 s	2.396 s	21.961 s
float (precision = 5)	3.140 s	3.732 s	3.632 s
float (precision = 10)	4.288 s	6.508 s	6.312 s

3.0 FORMAL VERIFICATION OF LINEAR PROGRAMS

An important part of the Flyspeck project [Hal12b] consists of more than 50,000 linear programs (with about 1000 variables and constraints each). A bound of each linear program needs to be formally verified.

We present a tool for proving bounds of linear programs. The hard computational work is done using external software for solving linear programs. This software returns a special certificate which is used in the formal verification procedure. There are two main difficulties in this approach. One is the precision of the computer arithmetic. Usually, results of computer floating-point operations are not exact. Meanwhile, precise results are necessary for producing formal proofs. The second problem is the speed of the formal arithmetic.

Steven Obua in his thesis [Obu08] developed a tool for verifying a part of Flyspeck linear programs. His work is done in the Isabelle proof assistant. We have made two significant advances over this earlier work. First, we developed a tool for verifying bounds of general linear programs which verifies Flyspeck linear program much faster than Obua's program. Our tool is able to verify a single relatively large Flyspeck linear program in about 5 seconds. In Obua's work, the time of verification of a single linear program varies from 8.4 minutes up to 67 minutes. Second, our work is done in HOL Light so it is not required to translate results from one proof assistant into another.

Our work consists of two parts. The first part (Sections 3.1 and 3.2) describes a tool for verification of bounds of general linear programs. This work has been published [SH11]. The second part (Section 3.3) is verification of bounds of Flyspeck linear programs. Right now, we are able to verify most of Flyspeck linear programs formally but there are still missing result for remaining linear programs. Section 3.4 describes all necessary work to finish the formal verification. Section 3.5 contains performance test results for out tool.

3.1 VERIFICATION OF BOUNDS OF LINEAR PROGRAMS

Our goal is to prove inequalities in the form $\mathbf{c}^T\mathbf{x} \le K$ such that $\mathbf{A}\mathbf{x} \le \mathbf{b}$ and $\mathbf{l} \le \mathbf{x} \le \mathbf{u}$, where $\mathbf{c}$, $\mathbf{b}$, $\mathbf{l}$, $\mathbf{u}$ are given n-dimensional vectors, $\mathbf{x}$ is an n-dimensional vector of variables, K is a constant, and $\mathbf{A}$ is an $m \times n$ matrix. To solve this problem, we consider the following linear program

$$\text{maximize } \mathbf{c}^T\mathbf{x} \text{ subject to } \bar{\mathbf{A}}\mathbf{x} \le \bar{\mathbf{b}},\ \bar{\mathbf{A}} = \begin{pmatrix} \mathbf{A} \\ -\mathbf{I}_n \\ \mathbf{I}_n \end{pmatrix},\ \bar{\mathbf{b}} = \begin{pmatrix} \mathbf{b} \\ -\mathbf{l} \\ \mathbf{u} \end{pmatrix}.$$

Suppose that $M = \max \mathbf{c}^T\mathbf{x}$ is the solution to this linear program. We require that $M \le K$. In fact, for our method we need a strict inequality $M < K$ because we employ numerical methods which do not give exact solutions.

We do not want to solve the linear program given above using formal methods. Instead, we use general software for solving linear programs which produces a special certificate that can be used to formally verify the original upper bound. Consider a dual linear program

$$\text{minimize } \mathbf{y}^T\bar{\mathbf{b}} \text{ subject to } \mathbf{y}^T\bar{\mathbf{A}} = \mathbf{c}^T,\ \mathbf{y} \ge 0.$$

The general theory of linear programming asserts that if the primal linear program has an optimal solution, then the dual program also has an optimal solution such that $\min \mathbf{y}^T\bar{\mathbf{b}} = \max \mathbf{c}^T\mathbf{x} = M$. Suppose that we can find an optimal solution to the dual program, i.e., assume that we know $\mathbf{y}$ such that $\mathbf{y}^T\bar{\mathbf{b}} = M \le K$ and $\mathbf{y}^T\bar{\mathbf{A}} = \mathbf{c}^T$. Then we can formally verify the original inequality by doing the following computations in a formal way:

$$\mathbf{c}^T\mathbf{x} = (\mathbf{y}^T\bar{\mathbf{A}})\mathbf{x} = \mathbf{y}^T(\bar{\mathbf{A}}\mathbf{x}) \le \mathbf{y}^T\bar{\mathbf{b}} = M \le K.$$

Our algorithm can be split into two parts. In the first part, we compute a solution $\mathbf{y}$ to the dual problem. In the second part, we formally prove the initial inequality using the computed dual solution and doing all arithmetic operations in a formal way.

We impose additional constraints on the input data. We suppose that all coefficients and constants can be approximated by finite decimal numbers such that a solution of the approximated problem implies the original inequality. Consider a simple example. Suppose we need to prove the inequality $x - y \leq \sqrt{3}$ subject to $0 \leq x \leq \pi$ and $\sqrt{2} \leq y \leq 2$. In general, an approximation that loosens the domain and tightens the range implies the original inequality. For example, consider an approximation of proving $x - y \leq 1.732$, subject to $0 \leq x \leq 3.142$ and $1.414 \leq y \leq 2$. It is easy to see that if we can prove the approximated inequality, then the verification of the original inequality trivially follows. From now on, we assume that entries of $\bar{\mathbf{A}}$, $\bar{\mathbf{b}}$, $\mathbf{c}$, and the constant K are finite decimal numbers with at most p_1 decimal digits after the decimal point.

We need to find a vector $\mathbf{y}$ with the following properties:

$$\mathbf{y} \geq 0, \quad \mathbf{y}^T \bar{\mathbf{A}} = \mathbf{c}^T, \quad \mathbf{y}^T \bar{\mathbf{b}} \leq K.$$

Moreover, we require that all elements of $\mathbf{y}$ are finite decimal numbers.

In our work, we use `GLPK` (GNU Linear Programming Kit) software for solving linear programs [GLP]. The input of this program is a model file which describes a linear program in the AMPL modeling language [AMP]. `GLPK` automatically finds solutions of the primal and dual linear programs. We are interested in the dual solution only. Suppose $\mathbf{r}$ is a numerical solution to the dual problem. Take its decimal approximation $\mathbf{y}_1^{(p)}$ with p decimal digits after the decimal point. We have the following properties of $\mathbf{y}_1^{(p)}$:

$$\mathbf{y}_1^{(p)} \geq 0, \quad M \leq \bar{\mathbf{b}}^T \mathbf{y}_1^{(p)}, \quad \bar{\mathbf{A}}^T \mathbf{y}_1^{(p)} = \mathbf{c} + \mathbf{e}.$$

The vector $\mathbf{e}$ is the error term from numerical computation and decimal approximation.

We need to modify the numerical solution $\mathbf{y}_1^{(p)}$ to get $\mathbf{y}_2^{(p)}$ such that $\bar{\mathbf{A}}^T \mathbf{y}_2^{(p)} = \mathbf{c}$. Write $\mathbf{y}_1^{(p)} = (\mathbf{z}^T, \mathbf{v}^T, \mathbf{w}^T)^T$ where $\mathbf{z}$ is an m-dimensional vector, $\mathbf{v}$ and $\mathbf{w}$ are n-dimensional vectors. Define $\mathbf{y}_2^{(p)}$ as follows

$$\mathbf{y}_2^{(p)} = \begin{pmatrix} \mathbf{z} \\ \mathbf{v} + \mathbf{v}_e \\ \mathbf{w} + \mathbf{w}_e \end{pmatrix}, \quad \mathbf{v}_e = \frac{|\mathbf{e}| + \mathbf{e}}{2}, \quad \mathbf{w}_e = \frac{|\mathbf{e}| - \mathbf{e}}{2}.$$

In other words, if $e_i > 0$ (the i-th component of $\mathbf{e}$), then we add e_i to v_i, otherwise we add $-e_i$ to w_i. We obtain $\mathbf{y}_2^{(p)} \geq 0$. Moreover,

$$\bar{\mathbf{A}}^T \mathbf{y}_2^{(p)} = \mathbf{A}^T \mathbf{z} - (\mathbf{v} + \mathbf{v}_e) + (\mathbf{w} + \mathbf{w}_e) = \bar{\mathbf{A}}^T \mathbf{y}_1^{(p)} - \mathbf{e} = \mathbf{c}.$$

Note that elements of $\mathbf{y}_2^{(p)}$ are finite decimal numbers. Indeed, $\mathbf{y}_2^{(p)}$ is obtained by adding some components of the error vector $\mathbf{e} = \bar{\mathbf{A}}^T \mathbf{y}_1^{(p)} - \mathbf{c}$ to the vector $\mathbf{y}_1^{(p)}$, and all components of $\bar{\mathbf{A}}$, $\mathbf{c}$, and $\mathbf{y}_1^{(p)}$ are finite decimal numbers.

If $\bar{\mathbf{b}}^T \mathbf{y}_2^{(p)} \leq K$, then we are done. Otherwise, we need to find $\mathbf{y}_1^{(p+1)}$ using higher precision decimal approximation of $\mathbf{r}$ and consider $\mathbf{y}_2^{(p+1)}$. Assuming that the numerical solution $\mathbf{r}$ can be computed with arbitrary precision and that $M < K$, we eventually get $\bar{\mathbf{b}}^T \mathbf{y}_2^{(s)} \leq K$.

From the computational point of view, we are interested in finding an approximation of the dual solution such that its components have as few decimal digits as possible (formal arithmetic on small numbers works faster). We start from a small value of p_0 (we choose $p_0 = 3$ for Flyspeck linear programs) and construct $\mathbf{y}_2^{(p_0)}, \mathbf{y}_2^{(p_0+1)}, \ldots$ until we get $\bar{\mathbf{b}}^T \mathbf{y}_2^{(p_0+i)} \leq K$.

We implemented a program in C# which takes an explicit linear program in the `lp` format [LPS] generated automatically from the input model file and a dual solution obtained with `GLPK`. The program returns an approximate dual solution (with as low precision as possible) which then can be used in the formal verification step. The program algorithm is the following. It reads in the input linear program and the corresponding solution and then approximates all inequalities and constants with some initial precision. Then a bound of the linear program is verified using exact decimal arithmetic (type `decimal` in C#). If the verification fails, a higher precision is considered. When the verification is successful than a special HOL Light file is generated which contains all input information for the formal verification procedure. The current implementation of our program does not work with arbitrary precision arithmetic, so it could fail on some linear programs. It should be not a problem for many practical cases because the `decimal` type in C# can exactly represent 28-digit decimal numbers [DEC] (for instance, we need at most 6 decimal digits for proving Flyspeck linear programs).

3.2 FORMAL VERIFICATION

Our aim is to verify the inequality $\mathbf{c}^T\mathbf{x} \le K$ using the computed dual solution approximation $\mathbf{y}^T = (\mathbf{z}^T, \mathbf{v}^T, \mathbf{w}^T)$ (we write $\mathbf{y}$ for the approximation $\mathbf{y}_2^{(s)}$, computation of which is described in the previous section):

$$\mathbf{c}^T\mathbf{x} = \mathbf{z}^T(\mathbf{A}\mathbf{x}) - \mathbf{v}^T\mathbf{x} + \mathbf{w}^T\mathbf{x} = \mathbf{y}^T\bar{\mathbf{A}}\mathbf{x} \le \mathbf{y}^T\bar{\mathbf{b}} \le K.$$

Here $\mathbf{x}$ is an n-dimensional vector of variables, $\mathbf{x} = (x_1, \ldots, x_n)$. We need to verify two results using formal arithmetic: $\mathbf{y}^T\bar{\mathbf{A}}\mathbf{x} = \mathbf{c}^T\mathbf{x}$ and $\mathbf{y}^T\bar{\mathbf{b}} \le K$.

The computation of $\mathbf{y}^T\bar{\mathbf{b}}$ is a straightforward application of formal arithmetic operations. $\mathbf{y}^T\bar{\mathbf{A}}\mathbf{x}$ can be computed efficiently. Usually, the matrix $\bar{\mathbf{A}}$ is sparse, so it makes no sense to do a complete matrix multiplication in order to compute $\mathbf{y}^T\bar{\mathbf{A}}\mathbf{x}$. The i-th constraint inequality can be written in the form

$$\sum_{j \in I_i} a_{ij}x_j \le b_i,$$

where I_i is the set of indices such that $a_{ij} \ne 0$ for $j \in I_i$, and $a_{ij} = 0$ for $j \notin I_i$. Also we have $2n$ inequalities for bounds of $\mathbf{x}$: $l_i \le x_i \le u_i$.

Define a special function in HOL Light for representing the left hand side of a constraint inequality

$$\mathrm{linf}([]) = 0 \ \wedge \ \mathrm{linf}\big((a_1, x_1) :: t\big) = a_1x_1 + \mathrm{linf}(t)$$

The function linf has the type `:(real#real)list->real`. It means that linf takes a list of pairs of real-valued elements and returns a real value. The function linf is defined recursively: we define its value on the empty list [], and we specify how linf can be computed on a k-element list using its value on a $(k-1)$-element list.

We suppose that all constraints and bounds of variables are theorems in HOL Light and each such theorem has the form

$$\vdash \alpha_1x_1 + \ldots + \alpha_kx_k \le \beta.$$

We have a conversion which transforms $\alpha_1x_1 + \ldots + \alpha_kx_k$ into the corresponding function $\mathrm{linf}[(\alpha_1, x_1); \ldots ; (\alpha_k, x_k)]$. Also, variables x_i may have different names (like $var1$, $x4$, $y34$, etc.), and after the conversion into linf, all elements in the list will be sorted using some fixed ordering on the names of variables (usually, it is a lexicographic ordering). For efficiency, it is important to assume that the variables in the objective function $\mathbf{c}^T\mathbf{x}$ (i.e., variables for which $c_i \neq 0$) are the last ones in the fixed ordering (we can always satisfy this assumption by renaming the variables).

First of all, we need to multiply each inequality by the corresponding value $0 \leq y_i$. It is a straightforward computation based on the following easy theorem

$$\vdash c \times \operatorname{linf}\big([(a_1, x_1); \ldots; (a_k, x_k)]\big) = \operatorname{linf}\big([(c \times a_1, x_1); \ldots (c \times a_k, x_k)]\big).$$

Note that we can completely ignore inequalities for which $y_i = 0$ because they do not contribute to the sum which we want to compute.

The main step is computation of the sum of two linear functions. Suppose we have two linear functions $\operatorname{linf}[(a, x_i); t_1]$ and $\operatorname{linf}[(b, x_j); t_2]$ (t_1 and t_2 denote tails of the lists of pairs). Depending on the relation between x_i and x_j (i.e., we compare the names of variables), we need to consider three cases: $x_i \equiv x_j$ (the same variables), $x_i \prec x_j$ (in the fixed ordering), or $x_i \succ x_j$. In the first case, we apply the following theorem

$$\vdash \operatorname{linf}\big((a, x) :: t_1\big) + \operatorname{linf}\big((b, x) :: t_2\big) = (a + b) \times x + \big(\operatorname{linf}(t_1) + \operatorname{linf}(t_2)\big).$$

In the second case, we have the theorem

$$\vdash \operatorname{linf}\big((a, x_i) :: t_1\big) + \operatorname{linf}\big((b, x_j) :: t_2\big) = a \times x_i + \Big(\operatorname{linf}(t_1) + \operatorname{linf}\big((b, x_j) :: t_2\big)\Big).$$

In the third case, the result is analogous to the second case. After applying one of these theorems, we recursively compute the expression in the parentheses and $a + b$ (if necessary). Then we can apply the following simple result and finish the computation of the sum

$$\vdash a \times x + \operatorname{linf}(t) = \operatorname{linf}\big((a, x) :: t\big).$$

Moreover, if the variables in both summands are ordered, then the variables in the result will be ordered.

After adding all inequalities for constraints, we get the inequality $\mathbf{z}^T\mathbf{A}\mathbf{x} \leq \mathbf{z}^T\mathbf{b}$ where the left hand side is computed in terms of linf. Now we need to find the sum of this inequality and inequalities for boundaries (multiplied by the corresponding coefficients). We do not transform boundary inequalities into the linf representation. Each boundary inequality has one of two forms

$$\vdash -x_i \le -l_i \quad \text{or} \quad \vdash x_i \le u_i.$$

Again, we need to multiply these inequality by the corresponding element of the dual solution vector $\mathbf{y} = (\mathbf{z}^T, \mathbf{v}^T, \mathbf{w}^T)^T$ and get

$$\vdash -v_i \times x_i \le v_i \times -l_i \quad \text{or} \quad \vdash w_i \times x_i \le w_i \times u_i.$$

If $v_i = 0$ or $w_i = 0$, then we can ignore the corresponding inequality. If for some x_i we have both boundary inequalities (lower and upper bounds) with non-zero coefficients, then find the sum of two such inequalities. After that, we get one inequality of the form $\vdash r_i x_i \le d_i$ for each variable x_i.

Before finding the sum of the inequality $\mathbf{z}^T\mathbf{A}\mathbf{x} \le \mathbf{z}^T\mathbf{b}$ and the boundary inequalities, we sort boundary inequalities using the same ordering we used for sorting variables in linf. We assumed that the variables in the objective function $\mathbf{c}^T\mathbf{x}$ are the last ones in our ordering. Let $c_i = 0$ for all $i \le n_0$. Suppose that we want to find the sum of $\vdash r_1 x_1 \le d_1$ and $\vdash \operatorname{linf}\big((a_1, x_1) :: t\big) \le s$. We know that $c_1 = 0$, so the first term $a_1 x_1$ in the linear function and $r_1 x_1$ must cancel each other, hence we have $a_1 = -r_1$. The formal sum can be found using the following result

$$\vdash a \times x + \operatorname{linf}\big((-a, x) :: t\big) = \operatorname{linf}(t).$$

Hence, we can efficiently compute the sum of all boundary inequalities for $i = 1, \ldots, n_0$. For the last $n - n_0$ variables, we have non-vanishing terms and the sum can be found in the standard way. Practically, the number $n - n_0$ is small compared to n, so most of computations are done in the efficient way.

At last, we get the inequality

$$\vdash \operatorname{linf}\big([(c_{n_0+1}, x_{n_0+1}); \ldots; (c_n, x_n)]\big) \le M',$$

where $M' = \mathbf{y}^T\mathbf{b} \le K$. It is left to prove that $M' \le K$. This can be done using standard HOL Light procedures because we need to perform this operation only once for each linear program.

Formal arithmetic operations on integers are considerably faster than operations on rational (decimal, floating-point) numbers. We want to perform all formal computations using integer numbers only. We have the following numerical values: entries of $\bar{\mathbf{A}}$, $\bar{\mathbf{b}}$, $\mathbf{c}$, $\mathbf{y}$, and the constant K. The input data $\bar{\mathbf{A}}$, $\bar{\mathbf{b}}$, $\mathbf{c}$, and K can be approximated by finite decimal numbers with at most p_1 decimal digits after the decimal point. The dual solution $\mathbf{y}$ is constructed in such a way that all its elements have at most p_2 decimal digits after the decimal point.

We modify the main step of the algorithm as follows. Compute

$$(10^{p_1+p_2}\mathbf{c}^T)\mathbf{x} = (10^{p_2}\mathbf{y}^T)\left(10^{p_1}\bar{\mathbf{A}}\right)\mathbf{x} \le (10^{p_2}\mathbf{y}^T)(10^{p_1}\bar{\mathbf{b}}) \le 10^{p_1+p_2}K.$$

It is clear, that the computations above can be done using integer numbers only. In the last step, we divide both sides by $10^{p_1+p_2}$ (using formal rational arithmetic only one time) and obtain the main inequality.

3.3 FLYSPECK LINEAR PROGRAMS

To describe Flyspeck linear programs, we need to define some important objects from the proof of the Kepler conjecture [Hal12a]. These objects are hypermaps and fans. Fans are geometric objects which can be constructed for any potential counterexample to the Kepler conjecture. Hypermaps are purely combinatorial objects. Each fan has an associated hypermap which encodes combinatorial information of the fan. Fans corresponding to counterexamples have hypermaps with special properties. These hypermaps are called restricted hypermaps and it is possible to get a complete finite list of all restricted hypermaps with a given number of darts. There are many relations between components of a fan (sums of angles, distances between points, etc.). All nonlinear functions can be replaced with variables in these relations, and then one gets a linear program corresponding to the given fan. The linear programs contain specific geometric information, but only information that can be encoded linearly into the combinatorics of the associated hypermap. Thus, for each restricted hypermap, one gets a linear program. And it can be proved that all these linear programs are not feasible, hence there are no counterexamples to the Kepler conjecture.

Definition. *A* hypermap *is a finite set D together with three functions $e, n, f : D \to D$ such that $e \circ n \circ f = I_D$. The elements of D are called* darts. *The functions e, n and f are called the* edge map*, the* node map*, and the* face map*, respectively. A* node *of a hypermap $H = (D, e, n, f)$ is the orbit of a dart $x \in D$ under n. A* face *is an orbit under f. An* edge *is an orbit under e.*

A hypermap with some special properties is called a restricted hypermap. The following theorem [Hal12a] tells that there are finitely many restricted hypermaps and it is possible to generate all of them.

Theorem. *There exists an algorithm which generates all restricted hypermaps with a given number of darts up to isomorphism.*

Definition. *Let $H_1 = (D_1, e_1, n_1, f_1)$ and $H_2 = (D_2, e_2, n_2, f_2)$ be hypermaps. A bijective function $r : D_1 \to D_2$ is an isomorphism between H_1 and H_2 if $e_2 = r \circ e_1$, $n_2 = r \circ n_1$, and $f_2 = r \circ f_1$.*

We don't give a formal definition of a fan. It is enough to know that each fan is defined by a pair (V, E) consisting of a set $V \subset \mathbb{R}^3$ and a set E of unordered pairs of distinct elements of V. The set of darts of the associated hypermap is a subset of V^2. To understand fans and associated hypermaps better, consider the following connection between fans and polyhedra. It is proved that each bounded polyhedron P in $\mathbb{R}^3$ with nonempty interior defines a fan (V_p, E_p) where V_p are vertices of P and E_p are edges of P. Darts of the associated hypermap H_p are directed edges of P. There are one-to-one correspondences between facets of P and faces of H_p, edges of P and edges of H_p, and between vertices of P and nodes of H_p.

A complete list of restricted hypermaps with a given number of darts can be algorithmically generated [Hal12a, NBS06]. The algorithm which computes this list encodes each hypermap as a list of lists of numbers. That is, for a hypermap G in the algorithmically generated list, we have $G = \mathrm{HYP}([l^1; l^2; \ldots; l^k])$ where $l^i = [d^i_1; \ldots; d^i_{k_i}]$. The function HYP converts lists of lists of numbers into hypermaps. The definition of this function is the following.

Definition. *Suppose* $H^l = \mathrm{HYP}([l^1; l^2; \ldots; l^k]) = (D^l, e^l, n^l, f^l)$*. Then*

$$D^l = \bigcup_{j=1}^{k} \{(d_1^j, d_2^j), (d_2^j, d_3^j) \ldots, (d_{k_j-1}^j, d_{k_j}^j), (d_{k_j}^j, d_1^j)\},$$

$$\begin{aligned} e^l(n,m) &= (m,n), \\ f^l(d_i^j, d_{i+1}^j) &= (d_{i+1}^j, d_{i+2}^j), \\ n^l(d_i^j, d_{i+1}^j) &= (e^l \circ (f^l)^{-1})(d_i^j, d_{i+1}^j) = (d_i^j, d_{i-1}^j). \end{aligned}$$

All indices are computed modulo sizes of corresponding lists.

Each list l^j corresponds to a face. Each number d_i^j defines a node (as a set of all darts with the first component d_i^j). Darts D^l are pairs of consecutive numbers in the same face (in the cyclic order). For instance, if we have a list $[[1; 2]]$, then $D^l = \{(1, 2), (2, 1)\}$, there are two nodes $\{(1, 2)\}$ and $\{(2, 1)\}$ (corresponding to numbers 1 and 2), one face $\{(1, 2), (2, 1)\}$ (corresponding to $[1; 2]$).

We have special formal procedures which compute all elements of a hypermap defined by a list of lists of numbers.

Let (V, E) be a fan corresponding to a counterexample. Then the associated hypermap $H = (D, e, n, f)$ is restricted and has some special property (for instance, it has only 13, 14, or 15 nodes and the number of its faces and darts is bounded by some constant). This hypermap belongs to a class of so called tame hypermaps. There is a hypermap H^l in the list of restricted hypermaps such that H^l and H are isomorphic (denote the isomorphism by $g : D^l \to D$).

Constraints of the linear program associated with the hypermap H^l have the following form

$$\forall x \in X, \sum_{i \in S(x)} c_i^{(x)} v_i^{(x)} \leq b^{(x)}.$$

Here, X is a set of some elements of the hypermap (it could be a set of all darts, a set of faces, a particular face, etc.), $c_i^{(x)}$ and $v_i^{(x)}$ are coefficients and variables indexed by x and by i iterating over a set $S(x)$ which depends on x. Each linear constraint corresponds to a relation between elements of a fan where all nonlinear functions are replaced with variables indexed by the arguments of nonlinear functions (these nonlinear functions are computed on hypermap elements).

Here are several examples of fan relations

$$\text{lnsum: } \sum_{v \in \text{nodes}(H)} L(v) > 12,$$

$$\text{azimsum: } \forall v \in \text{nodes}(H),\ \sum_{(\mathbf{x},\mathbf{y}) \in v} \text{azim}(\mathbf{x},\mathbf{y}) = 2\pi,$$

$$\text{rha: } \forall (\mathbf{x},\mathbf{y}) \in D(H),\ \text{rhazim}(\mathbf{x},\mathbf{y}) \geq \text{azim}(\mathbf{x},\mathbf{y}).$$

Recall, that each node is a set of darts and each dart is a pair of vectors (since we consider a hypermap associated with a fan). Functions L, azim, and rhazim are defined as follows [Hal12a]:

$$L(\mathbf{x},\mathbf{y}) = \begin{cases} \dfrac{2.52 - \|\mathbf{x}\|}{2.52 - 2} & \text{if } \|\mathbf{x}\| \leq 2.52, \\ 0 & \text{otherwise,} \end{cases}$$

$$\text{azim}(\mathbf{x},\mathbf{y}) = \text{dih}\big(\{\mathbf{0},\mathbf{x}\},\{\mathbf{y},\sigma(\mathbf{x},\mathbf{y})\}\big),$$

$$\text{rhazim}(\mathbf{x},\mathbf{y}) = \text{azim}(\mathbf{x},\mathbf{y})\left(1 + \frac{3\arccos(1/3) - \pi}{\pi}\big(1 - L(\mathbf{x},\mathbf{y})\big)\right).$$

(Here, $\sigma(\mathbf{x},\mathbf{y})$ is the second component of $n(\mathbf{x},\mathbf{y})$ and $\text{dih}\big(\{\mathbf{0},\mathbf{x}\},\{\mathbf{y},\sigma(\mathbf{x},\mathbf{y})\}\big)$ is the angle between half-planes defined by points $\{\mathbf{0},\mathbf{x},\mathbf{y}\}$ and $\{\mathbf{0},\mathbf{x},\sigma(\mathbf{x},\mathbf{y})\}$.)

We replace these functions with variables L_v, azim_d, and rhazim_d indexed over nodes v and darts $d \in D^l$ and get the corresponding linear constraints for the hypermap H^l:

$$\text{lnsum: } \sum_{v \in \text{nodes}(H^l)} L_v > 12,$$

$$\text{azimsum: } \forall v \in \text{nodes}(H^l),\ \sum_{d \in v} \text{azim}_d = 2\pi,$$

$$\text{rha: } \forall d \in D^l,\ \text{rhazim}_d \geq \text{azim}_d.$$

All these constraints are collected in an AMPL [AMP] model file. AMPL is a powerful algebraic modeling language for linear and nonlinear optimization problems. An excerpt of the model file for Flyspeck linear programs is given below.

```
lnsum_def: sum{i in node} ln[i]  = lnsum;

azim_sum{i in node}:  sum {(i,j) in dart} azim[i,j] = 2.0*pi;

RHA{(i,j) in dart}: rhazim[i,j] >= azim[i,j]*1.0;
```

This excerpt shows how constraints of linear programs are encoded in the model file. Each constraint consists of a label and an inequality or equality. Labels can be indexed by elements of sets which are also defined in the model file (e.g., the label `azim_sum` has index `i` which assumes values from the set `node`).

Each variable in the linear program is indexed by a hypermap element and each variable has specific bounds. For instance, here are several variables defined in the AMPL model file

```
var azim{dart} >= 0, <= pi;

var ln{node} >= 0, <= 1;

var sol{face} >= 0, <= 4.0*pi;
```

The total number of variables in a Flyspeck linear program is more than 700. The objective of each linear program is to maximize the function `sum{i in node} ln[i]`, i.e.,

$$\sum_{v \in \text{nodes}(H^l)} L_v.$$

For a counterexample, the value of this function should be greater than 12. If the maximum of a linear program is less than 12, it will mean that the corresponding packing is not a counterexample. It is necessary to check that the maximum is less than 12 for all potential counterexamples, that is, for all restricted hypermaps (with specific number of darts).

The model file has parameters. Parameters of the model file are sets of elements of a hypermap. All parameters are informally computed from a given hypermap H^l. Note that the internal representation of hypermap elements in the model file is not the same as we defined above for formal proofs in HOL Light. For instance, nodes correspond to just one number. Nevertheless, there is one-to-one correspondence between hypermap elements in the model file and hypermap elements defined in HOL Light.

To generate a particular linear program, it is necessary to create a parameter file and run the `GLPK` solver [GLP] on the model file with the given parameter file. `GLPK` generates an explicit linear program and saves it in an `lp` file [LPS]. Also, `GLPK` produces a solution file.

The explicit linear program and its solution are not enough to completely formally verify the linear program. We need to generate theorems corresponding to constraints of linear programs. These theorems must be based on relations between elements of a fan. Consider how it is done on an example.

We have the following theorem

$$\vdash \forall v \in \operatorname{nodes}(H),\ \sum_{d\in v} \operatorname{azim}(d) = 2\pi.$$

Here H is the hypermap of a fan. We are working with hypermaps of lists. So the first step is to apply an isomorphism g and get

$$\operatorname{iso}(g, H^l, H) \vdash \forall v \in \operatorname{nodes}(H^l),\ \sum_{d\in v} \operatorname{azim}\bigl(g(d)\bigr) = 2\pi.$$

(There are formal results which justify this conversion.) Now, we replace the nonlinear function $(\operatorname{azim} \circ g)(d)$ with a variable azim_d indexed over darts and transform the equality into two inequalities (it is required only for equality constraints). We get

$$\forall v,\ v \in \operatorname{nodes}(H^l) \implies \sum_{d\in v} \operatorname{azim}_d \le 2\pi,$$
$$\forall v,\ v \in \operatorname{nodes}(H^l) \implies \sum_{d\in v} \operatorname{azim}_d \ge 2\pi.$$

(Note that these are terms, not theorems; theorems are obtained by replacing azim_d with $(\operatorname{azim} \circ g)(d)$ and by adding the isomorphism assumption.) It is time to get rid of 2π. Approximate each inequality with decimal numbers with a fixed number of decimal digits after the decimal point. If the approximation is 2 decimal digits, we get

$$\forall v,\ v \in \operatorname{nodes}(H^l) \implies \sum_{d\in v} \operatorname{azim}_d \le 2\pi \quad \vdash \forall v,\ v \in \operatorname{nodes}(H^l) \implies \sum_{d\in v} \operatorname{azim}_d \le 6.29,$$
$$\forall v,\ v \in \operatorname{nodes}(H^l) \implies \sum_{d\in v} \operatorname{azim}_d \ge 2\pi \quad \vdash \forall v,\ v \in \operatorname{nodes}(H^l) \implies \sum_{d\in v} \operatorname{azim}_d \ge 6.28.$$

(These are theorems with premises.) It is left to convert all constants into integer numbers

$$\forall v,\ v \in \operatorname{nodes}(H^l) \implies \sum_{d\in v} \operatorname{azim}_d \le 2\pi \quad \vdash \forall v,\ v \in \operatorname{nodes}(H^l) \implies \sum_{d\in v} 100\operatorname{azim}_d \le 629,$$
$$\forall v,\ v \in \operatorname{nodes}(H^l) \implies \sum_{d\in v} \operatorname{azim}_d \ge 2\pi \quad \vdash \forall v,\ v \in \operatorname{nodes}(H^l) \implies \sum_{d\in v} 100\operatorname{azim}_d \ge 628.$$

This transformation procedure is completely automatic and it is executed for all relations between fan elements which appear in the model file. Transformation results are stored in hash tables where keys are labels of the corresponding inequalities plus precision constants (we generate several versions of the same inequality for different precisions).

We use a modified C# program to produce a special input file for the verification procedure. For each constraint, this file contains a label of the corresponding inequality, indices of hypermap elements for which the inequality should be instantiated, and the corresponding dual solution coefficients (integer numbers). For instance, here is a line from the produced verification file

```
("azim_sum", [0; 1; 3; 6; 10; 11; 12], [87; 409; 409; 246; 139; 139; 246]);
```

Here, `"azim_sum"` refers to inequalities corresponding to the theorem

$$\vdash \forall v \in \mathrm{nodes}(H),\ \sum_{d \in v} \mathrm{azim}(d) = 2\pi.$$

The list $[0; 1; 3; 6; 10; 11; 12]$ gives indices of nodes for which the inequality must be used. Each node is a set of darts. We compute all such sets and save them in a list in a particular order. The indices in the verification file refer to elements of this list. The list $[87; 409; 409; 246; 139; 139; 246]$ contains dual solution coefficients.

An automatic verification procedure generates all necessary inequality theorems from the verification file. First of all, it formally computes sets of hypermap elements (e.g., faces, nodes, darts). Then, for a given inequality label, it finds a general prepared inequality and instantiates it using hypermap elements corresponding to indices in the verification file. Moreover, each verification file contains the precision constant, and the verification procedure finds the prepared inequality with the given precision of coefficients. All inequalities are in the form

$$\text{general inequality} \vdash \text{specific inequality}.$$

Using these inequality theorems, we verify the given linear program with the procedure described in Section 3.2. The result will be in the form

$$\text{general inequalities} \vdash \sum_{v \in \mathrm{nodes}(H^l)} L_v \le 12.$$

We replace variables with corresponding nonlinear functions and add the isomorphism assumption. After this step, all general inequalities become theorems and so they can be eliminated. And hence the final result will be

$$\vdash \mathrm{iso}(g, H^l, H) \implies \sum_{v \in \mathrm{nodes}(H^l)} L(v) \leq 12.$$

For a counterexample, we have the isomorphism theorem $\vdash \mathrm{iso}(g, H^l, H)$ and the opposite inequality theorem $\vdash \sum_{v \in \mathrm{nodes}(H^l)} L(v) > 12$. We obtain a contradiction which eliminates potential counterexamples for the hypermap H^l.

3.4 FUTURE WORK

Not all Flyspeck linear program can be verified directly. Some of them must be split into new linear programs by introducing new constraints based on properties of hypermaps. The simplest example of splitting is a restriction of the perimeter of a triangular face (i.e., face with 3 darts). The following constraints are given in the model file

```
yy1 {(i,j) in dart_std3_big}:     y4[i,j]+y5[i,j]+y6[i,j] >= 6.25;
yy2 {(i,j) in dart_std3_small}:   y4[i,j]+y5[i,j]+y6[i,j] <= 6.25;
```

Variables `y4`, `y5`, and `y6` correspond to lengths of 3 edges of a face. Sets `dart_std3_big` and `dart_std3_small` are sets of big and small triangular faces. These sets are parameters of the model file. For each hypermap, we can consider two cases: a fixed triangular face is small or it is big. Each case produces different values of input parameters for the model file. If in both cases the liner program bound is less than 12, then we conclude that the same is true for the original hypermap (without restrictions on the perimeter of a fixed face).

There are other more complex cases: one can divide a face with more than 4 darts into two faces by adding a diagonal. Each subdivision will yield additional constraints which help to prove that bounds of new linear programs are less than 12.

3.5 TESTS

Table 6 contains performance test results for several Flyspeck linear programs. All tests were performed on Intel Core i5, 2.67GHz running Ubuntu 9.10 inside Virtual Box 4.2.0 on a Windows 7 host; the Ocaml version was 3.09.3; the base of natural number arithmetic was $b = 200$; the caching was turned on.

Table 6: Performance results for verification of linear program bounds

Linear program ID	# variables	# constraints	verification time (s)
118343205068	833	994	4.350
118760185161	807	869	3.758
119040238600	803	864	3.757
122526068934	783	824	3.638
123040027899	1002	1074	5.148
125719999821	693	824	3.210
147671934133	715	824	3.337
156401568298	924	1034	4.723
156615503428	826	864	4.071
158856256118	808	1034	4.329
165950391005	808	909	4.082
168156828154	834	909	4.165
17272290668	696	864	3.298
195482381558	689	784	3.161
196021155893	648	784	3.203
206084941231	720	784	3.309
211626865969	738	824	3.664
219955817888	770	869	3.867
245859035526	886	994	4.465
25168582633	700	784	3.318
30500231120	625	869	3.252
63626063287	845	954	4.311
69964410750	718	824	3.578
74394196986	747	954	4.020
91057093091	746	869	3.731

4.0 NONLINEAR INEQUALITIES

In this chapter, we present a tool for formal verification of nonlinear inequalities in HOL Light. Our tool can verify multivariate nonlinear inequalities on rectangular domains. The tool can verify both polynomial and non-polynomial inequalities. The verification technique is based on interval arithmetic with Taylor approximations. There is an existing tool in the PVS proof assistant which uses a similar technique [DLM09] but this tool works only with univariate functions.

There exist other formal methods for verification of nonlinear inequalities. First of all, general quantifier elimination procedures may be used to solve some polynomial inequalities [Tar51, Col75, MH05]. Another method for proving polynomial inequalities is known as sums-of-squares (SOS) method [Har07]. The idea of this method is to find a representation of a given polynomial $p(x)$ as a sum of squares of other polynomials. Then, the inequality $p(x) \geq 0$ can be immediately proved. One advantage of the SOS method is that it can verify inequalities on unbounded domains. A sums-of-squares representation of $p(x)$ can be found with informal procedures and then the formal verification is just a verification of an equality $p(x) = \sum p_i(x)^2$. For general multivariate polynomials, sums-of-squares representations are not easy to find. Even more, for some polynomial inequalities, it does not exist (see [Har07]).

A tool called MetiTarski [AP08, Pau12] is capable to verify multivariate non-polynomial inequalities on unbounded domains. It approximates non-polynomial functions by suitable polynomial bounds and then applies quantifier elimination procedures for resulting polynomials.

The Bernstein polynomial technique [Zum08] allows to verify multivariate polynomial inequalities. Each polynomial can be written as a sum of polynomials in the Bernstein polynomial basis. Coefficients of this representation give bounds of the polynomial itself. A complete formal implementation of this method is done in PVS [MN12]. Non-polynomial inequalities must be first converted into polynomial inequalities by finding polynomial bounds. One way to find polynomial bounds is to use Taylor model approximations [Zum06]. R. Zumkeller's thesis describes this method in details [Zum08]. He also implemented an informal global optimization tool based on Bernstein polynomials [Zum09] in Haskell [Has].

Methods based on quantifier elimination procedures do not scale well when the number of variables grows and when inequalities become more complicated. The Bernstein polynomial technique works well for polynomial inequalities but does not show very good results for inequalities involving special functions in high dimensions. Most of Flyspeck nonlinear inequalities are 6 variable non-polynomial inequalities. Thomas Hales implemented an informal verification procedure based on interval arithmetic with Taylor approximations which can completely verify all Flyspeck nonlinear inequalities [Hal03]. This chapter presents a formal implementation of this procedure.

4.1 VERIFICATION OF NONLINEAR INEQUALITIES

4.1.1 Nonlinear inequalities and interval Taylor approximations

Consider the problem: prove that

$$\forall \mathbf{x} \in \mathbb{R}^n, \mathbf{x} \in D \implies f(\mathbf{x}) < 0.$$

D is assumed to be a rectangle given by $D = \{(x_1, \ldots, x_n) \mid a_i \le x_i \le b_i\} = [\mathbf{a}, \mathbf{b}]$. We also assume that $f(\mathbf{x})$ is twice continuously differentiable in an open domain $U \supset D$.

One way to solve the problem is to consider a finite partition of $D = \bigcup_j D^j$ such that each D^j is rectangular. Also, we assume that $\bar{f}(D^j) < 0$ where $\bar{f}$ is an interval approximation of f (that is, $\bar{f}(D^j)$ is the interval corresponding to the interval evaluation of $f(x_1, \ldots, x_n)$ for input intervals $x_i \in [a_i^j, b_i^j]$; clearly, $\bar{f}(D) < 0 \implies f(D) < 0$). It is easy to see that such a partition always exists if f is continuous, $f(D) < 0$, and f can be arbitrary well approximated by $\bar{f}$ on sufficiently small domains. (It follows by the compactness argument: for each point $x \in D$ there is a small rectangle D^j such that $x \in \text{interior}(\mathrm{D^j})$ and $\bar{f}(D^j) < 0$; D is compact, so there are finitely many rectangles D^j such that $D = \bigcup_j D^j$.)

The main difficulty is finding a suitable partition $\{D^j\}$. The easiest way is the following. Let $D^0 = D$ and compute $\bar{f}(D^0)$. If this value is less than 0 (in the interval sense), then we are done. Otherwise divide D^0 into two regions $D^0 = D^1_1 \cup D^1_2$. Then repeat the procedure for regions with upper index 1. In general, either $\bar{f}(D^k_j) < 0$ or we get $D^k_j = D^{k+1}_{2j-1} \cup D^{k+1}_{2j}$. If we divide each region such that sizes of new regions become arbitrary small in all dimensions, then the process will eventually stop and a suitable partition of D will be found. An easy way to achieve this goal is to divide each region in half along the coordinate for which its size is maximal, i.e., if $D^k_j = \{a_i \le x_i \le b_i\} = [\mathbf{a}, \mathbf{b}]$ and $b_m - a_m = \max_i\{b_i - a_i\}$, then set $D^{(k+1)_{2j-1}} = [\mathbf{a}, \mathbf{b}^{(m,y)}]$ and $D^{(k+1)}_{2j} = [\mathbf{a}^{(m,y))}, \mathbf{b}]$. Here, $y = (a_m + b_m)/2$ and $\mathbf{a}^{(m,y)}$ equals to $\mathbf{a}$ with the m-th component replaced by y.

As the result of the procedure above, we get a finite set of subregions $S = \{D^k_i\}$ with the property: for each $D^k_i \in S$ either $\bar{f}(D^k_i) < 0$ or $D^k_i = D^{k+1}_{i_1} \cup D^{k+1}_{i_2}$. In the last case, the verification relies on a trivial theorem

$$D = D_1 \cup D_2 \ \wedge\ f(D_1) < 0 \ \wedge\ f(D_2) < 0 \implies f(D) < 0.$$

Interval arithmetic works for any continuous function (at least in theory where numerical errors are not considered) but it is not very efficient in general. This is due to the dependency problem when even a simple function could require a lot of subdivisions in order to get the result on the full domain. Even a trivial inequality $f(x) = x - x < 1$ will require subdivisions for the domain $x \in [0,1]$. Indeed, $\bar{f}([0,1]) = [0,1] - [0,1] = [-1,1]$. Of course, we can simplify $x - x = 0$ but it is not possible to do for a function $f(x) = x - \arctan(x)$ which has similar behaviour near 0. For this function, $\bar{f}([0,1]) = [0,1] - [0,\pi/4] = [-\pi/4, 1]$ and we don't get $f(x) < 1$. One way to decrease the dependency problem is to use Taylor approximations for computing bounds of f on a given domain D.

Fix $\mathbf{y} \in D = [\mathbf{a}, \mathbf{b}]$, then we can write

$$f(\mathbf{x}) = f(\mathbf{y}) + \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(\mathbf{y})(y_i - x_i) + \frac{1}{2}\sum_{i,j=1}^{n} \frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{p})(y_i - x_i)(y_j - x_j)$$

where $\mathbf{p} \in [\mathbf{a}, \mathbf{b}]$. Let $\mathbf{w} = \max\{\mathbf{y} - \mathbf{a}, \mathbf{b} - \mathbf{y}\}$ (all operations are componentwise). Suppose we have interval bounds for $f(\mathbf{y}) \in [f_0^l, f_0^u]$, $\frac{\partial f}{\partial x_i}(\mathbf{y}) \in [f_i^l, f_i^u]$ and $\frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{t}) \in [f_{ij}^l, f_{ij}^u]$ for all $\mathbf{t} \in D$. We can write

$$\forall \mathbf{x} \in D,\ f(\mathbf{x}) \le f(\mathbf{y}) + \sum_{i=1}^{n} \left| \frac{\partial f}{\partial x_i}(\mathbf{y}) \right| w_i + \frac{1}{2} \sum_{i,j=1}^{n} \left| \frac{\partial^2 f}{\partial x_i \partial x_j}(\xi) \right| w_i w_j$$
$$\le f_0^u + \sum_{i=1}^{n} \left| [f_i^l, f_i^u] \right| w_i + \frac{1}{2} \sum_{i,j=1}^{n} \left| [f_{ij}^l, f_{ij}^u] \right| w_i w_j.$$

Absolute values of intervals are defined by $|[a,b]| = \max\{-a, b\}$.

Let's see how well this approximation works on examples. Again, take $f(x) = x - x$ and $D = [0,1]$. We compute $f'(x) = 1 - 1 = 0$ and $f''(x) = 0$. Set $y = 0.5$ and $w = 0.5$. Suppose $\bar{f}(0.5) = [0.4, 0.6] - [0.4, 0.6] = [-0.2, 0.2]$ (we deliberately take a very poor interval approximation), then

$$\forall x \in [0,1],\ f(x) \le \bar{f}(0.5)^u + \sum_{i=1}^{1} 0 \times 0.5 + \sum_{i,j=1}^{1} 0 \times 0.5 \times 0.5 = 0.2 < 1.$$

In the same way, for $f(x) = x - \arctan x$ we get $f'(x) = 1 - \frac{1}{1+x^2}$, $f''(x) = \frac{-2x}{(1+x^2)^2}$. If $x \in [0,1]$, then $f''(x) \in [-2, 0] = [f_{11}^l, f_{11}^u]$ and hence $|f''(x)| \le 2$. We compute

$$\forall x \in [0,1],\ f(x) \le 0.04 + 0.21 \times 0.5 + 2 \times 0.5^3 \le 0.4.$$

We see that interval arithmetic with Taylor approximations works much better. Moreover, we don't need to abandon direct interval approximations completely: every time when we have to verify whether $f(D_i) < 0$ we can first find an interval approximation $\bar{f}(D_i)$ and then compute a Taylor approximation. If we don't get the inequality in both cases, then we subdivide the domain.

We will formally define Taylor interval approximations as interval bounds of f and its first-order partial derivatives at a fixed point $\mathbf{y} \in D$ and interval bounds of second-order partial derivatives for all $\mathbf{t} \in D$. Moreover, our formal definition will also include domain bounds $\mathbf{a}, \mathbf{b}$, the point $\mathbf{y}$ and the width parameters $\mathbf{w}$. See the end of Section 4.2 for the formal definition.

One simple trick which can be done with both interval and Taylor interval approximations is estimation of partial derivatives on a given domain. If it happens that $f_j(D_k) = \frac{\partial f}{\partial x_j}(D_k) \leq 0$ or $f_j(D_k) \geq 0$ then it will be immediately possible to restrict further verifications to the boundary of $D_k = [\mathbf{a}, \mathbf{b}]$. Indeed, if $f_j(D_k) \leq 0$ and $f(D_k|_{x_j=a_j}) < 0$ then $f(D_k) < 0$ since the function is decreasing along the j-th coordinate and its maximal value is attained at $x_j = a_j$. The same is true for increasing functions (consider $D_k|_{x_j=b_j}$). Moreover, if $\{x_j = a_j\}$ ($\{x_j = b_j\}$) is not on the boundary of the main domain D_k, then it is possible to completely ignore any further verifications for the region D_k. Indeed, if the restriction of D_k is not on the boundary of the original domain, then there is another subdomain D_j such that the restriction of D_k is a subset of D_j and the inequality is true on D_j. However, we need to be careful. Consider an example. Suppose $f(x) = -x^2 - 1$ and $D = [-1, 1]$. Assume that we have $D_1 = [-1, 0]$ and $D_2 = [0, 1]$. We get $f'(x) = -2x \geq 0$ on $[-1, 0]$. Hence, the function is increasing and we can consider the restricted domain $\{0\}$ which is not on the boundary of $[-1, 1]$. Also, $f'(x) = -2x \leq 0$ on $[0, 1]$ and we again get $\{0\}$ as the restriction of $[0, 1]$. If we don't continue verifications in both cases, then we will not be able to verify the inequality. In order to avoid this problem, we always check a strict inequality for decreasing functions, that is, we test if $f_j(\mathbf{x}) \geq 0$ or $f_j(\mathbf{x}) < 0$.

Another trick is to check convexity of a function before subdividing a domain D_k. If we need to subdivide D_k and find that $f_{jj}(D) = \frac{\partial^2 f}{\partial x_j \partial x_j}(D) \geq 0$, then it is enough to verify $f(D_k|_{x_j=a_j}) < 0$ and $f(D_k|_{x_j=b_j}) < 0$. By convexity of f (i.e., f attains its maximum on the boundary), we get $f(D_k) < 0$ from these two inequalities.

4.1.2 Solution certificate search procedure

An informal verification procedure based on the ideas presented above has been developed by Thomas Hales for informal verification of Flyspeck nonlinear inequalities [Hal03]. The starting point of our implementation of a formal procedure for verification of nonlinear inequalities is Hales' port of his original C++ program into OCaml. This OCaml program informally verifies a given nonlinear inequality on a rectangular domain by finding Taylor interval approximations and subdividing domains if necessary. The result of this program is just a boolean value: yes or no, the inequality true or false (there is the third option: verification could fail due to numerical instability or when subdomains become very small without any definite results).

We have modified the OCaml informal verification procedure such that it returns a partition of the original domain in a special tree-like structure which also contains all necessary information about verification steps for each subdomain. We call this structure a solution certificate for a given nonlinear inequality. The informal procedure is called the solution certificate search procedure.

A solution certificate is defined with the following OCaml record

```
type result_tree =
  | Result_false
  | Result_pass
  | Result_mono of mono_status list * result_tree
  | Result_glue of (int * bool * result_tree * result_tree)
  | Result_pass_mono of mono_status
  | Result_pass_ref of int
```

The record `mono_status` contains monotonicity information (i.e., whether some first-order partial derivative is negative or positive).

A simplified solution certificate search algorithm is given below

```
let search f dom =
  let taylor_inteval = {find Taylor interval approximation of f on dom}
  let bounds = {taylor_interval bounds}
  if bounds >= 0 then
    Result_false
  else if bounds < 0 then
    Result_pass
  else
    let d_bounds = {find bounds of partial derivatives from taylor_interval}
    let mono = {list of negative and positive partial derivatives}
```

```
if {mono is not empty} then
   let r_dom = {restrict dom using information from mono}
     Result_mono mono (search f r_dom)
else
  let dd_bounds = {find bounds of second partial derivatives on dom}
  if {the j-th second partial derivative is non-negative on dom} then
    let dom1, dom2 = {restrict dom along j}
    let c1 = search f dom1
    let c2 = search f dom2
      Result_glue (j, true, c1, c2)
  else
    let j = {find j such that b_i - a_i is maximal}
    let dom1, dom2 = {split dom along j}
    let c1 = search f dom1
    let c2 = search f dom2
      Result_glue (j, false, c1, c2)
```

If the inequality $f(x) < 0$ holds on D, then the algorithm (applied to f and D) will return a solution certificate which does not contain `Result_false` nodes (of course, the real algorithm could fail due to numerical instabilities and rounding errors). A solution certificate does not contain any explicit information about subdomains for which verification must be performed. All subdomains can be restored from a solution certificate and the initial domain D. For each `Result_glue(j, false, c1, c2)` node, it is necessary to split the domain in two halves along the j-th coordinate. The second argument is the convexity flag. If it is true, then the current domain must be restricted to its left and right boundaries along the j-th coordinate. For new subdomains, the node contains their solution certificates: `c1` and `c2`. The domain also has to be modified for `Result_mono` nodes. Each node of this type contains a list of indices and boolean parameters (packed in `mono_status` record) which indicate for which partial derivatives the monotonicity argument should be applied; boolean parameters determine if the corresponding partial derivatives are positive or negative.

The simplified algorithm never returns nodes of type `Result_pass_mono`. The real solution certificate search algorithm is a little more complicated. Every time when monotonicity argument is applied, it checks if the restricted domain is on the boundary of the original domain or not (the original domain is an argument of the algorithm). If the restricted domain is not on the boundary of the original domain, then `Result_pass_mono` will be returned.

If a solution certificate contains nodes of type `Result_pass_mono`, then it is necessary to transform such a certificate to get new certificates which can be formally verified. Indeed, suppose we have a `Result_pass_mono` node and the corresponding domain is D_k. `Result_pass_mono` requires to apply the monotonicity argument to D_k, that is, to restrict this domain to its boundary along some coordinate. But it doesn't contain any information on how to verify the inequality on the restricted subdomain. We can only claim that there is another subdomain D_j (corresponding to some other node of a solution certificate) such that the restriction of D_k is a subset of D_j. In other words, to verify the inequality on D_k, we first need to find D_j such that the restriction of D_k is a subset of D_j and such that the inequality can be verified on D_j. To solve this problem, we transform a given solution certificate into a list of solution certificates and subdomains for which these new solution certificates work. Each solution certificate in the list may refer to previous solution certificates with `Result_ref`. The last solution certificate in the list corresponds to the original domain. The transformation algorithm is the following

```
let transform certificate acc =
   let sub_certs = {find all maximal sub-certificates
                    which does not contain Result_pass_mono}
   if {sub_certs contains certificate} then
      {add certificate to acc and return acc}
   else
      let sub_certs = {remove certificates consisting of single
                        Result_ref from sub_certs}
      let paths = {find paths to sub-certificates in sub_cert}
      let _ = {add sub_certs and the corresponding paths to acc}
      let new_cert1 = {replace all sub_certs in certificate with references}
      let new_cert2 = {replace Result_pass_mono nodes in new_cert1 if
```

```
                    they can be verified using subdomains defined
                    by paths in acc}
         transform new_cert2 acc
```

This algorithm maintains a list `acc` of solution certificates which do not contain nodes of type `Result_pass_mono`. The list also contains paths to subdomains corresponding to certificates. Each path is a list of pairs and it can be used to construct the corresponding subdomain starting from the original domain. Each pair is one of `("l", i)`, `("r", i)`, `("ml", i)`, or `("mr", i)` where i is an index. `"l"` and `"r"` labels correspond to left and right subdomains after splitting. `"ml"` and `"mr"` correspond to left and right restricted subdomains. The index i specifies the coordinate along which the operation must be performed. When a reference node `Result_ref` is generated for a sub-certificate at the j-th position in the accumulator list `acc`, then the argument of `Result_ref` is j.

See Section 4.4 for an example of a solution certificate.

4.2 FORMAL THEORIES

The first step of developing a formal verification procedure is formalization of all necessary theories involving the multivariate Taylor theorem and related topics. All formalization was done with SSReflect mode in HOL Light (see Chapter 5).

Standard HOL Light libraries contain a formalization of Euclidean vector space [Har05] and define general Frechet derivatives and Jacobian matrices for working with first-order partial derivatives. Also, HOL Light contains a general univariate Taylor theorem.

First of all, first-order partial derivatives are defined explicitly

```
|- partial i f (x:real^N) =
         derivative (f o (\t. x + t % basis i)) (&0)
```

This definition corresponds to the following mathematical definition

$$\frac{\partial f}{\partial x_i}(\mathbf{x}) = \frac{d}{dt}\bigg|_{t=0} f(\mathbf{x} + t\mathbf{e}_i).$$

Relations between this definition and Frechet derivatives (and Jacobian matrices $\frac{\partial f}{\partial \mathbf{x}}$) are established and proved. In particular, we prove that

$$\mathrm{Diff}(f, \mathbf{y}) \implies \frac{\partial f}{\partial x_i}(\mathbf{y}) = \frac{\partial f}{\partial \mathbf{x}}(\mathbf{y})\mathbf{e}_i.$$

The notation $\mathrm{Diff}(f, \mathbf{y})$ means that f is differentiable at $\mathbf{y}$.

Another important result is formalization of the derivative of a composition of two functions $f : \mathbb{R}^n \to \mathbb{R}$ and $h : \mathbb{R} \to \mathbb{R}^n$.

$$\mathrm{Diff}(f, h(t)) \wedge \mathrm{Diff}(h, t) \implies (f \circ h)'(t) = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}\big(h(t)\big)h_i'(t).$$

The next result deals with monotonicity properties of partial derivatives

$$\begin{aligned}
&(\forall i, 1 \le i \le n \wedge i \ne j \implies u_i = z_i) \wedge u_j = x_j \wedge \mathrm{Diff}(f, [\mathbf{x}, \mathbf{z}]) \\
&\qquad \wedge \left(\forall \mathbf{y} \in [\mathbf{x}, \mathbf{z}] \implies 0 \le \frac{\partial f}{\partial x_j}(\mathbf{y})\right) \wedge (\forall \mathbf{y} \in [\mathbf{x}, \mathbf{u}] \implies l \le f(\mathbf{y})) \\
&\qquad \implies (\forall \mathbf{y} \in [\mathbf{x}, \mathbf{z}] \implies l \le f(\mathbf{y})).
\end{aligned}$$

The next step is to define second-order partial derivatives. The definition is simple

```
|- partial2 j i f = partial j (partial i f)
```

But this definition is not useful by itself. With this definition only, it is not possible to prove even that the second partial derivative of a sum is the sum of partial derivatives. It is required to define the notion of twice differentiable functions in order to prove results about second-order partial derivatives. The standard HOL Light definition of differentiability of a function at a point does not work here directly. We need to be able to define the first derivative in some neighbourhood of the point of interest before we can differentiate the function twice. Our approach is simple and is not most general: we insist that the function and its first partial derivatives are differentiable at some open neighbourhood of the given point. A more general approach might be useful if one wants to develop a complete theory of differentiability and partial derivatives. Our goal is to verify nonlinear inequalities where all functions are assumed to be sufficiently nice, so we don't need to follow the most general formalization path. Hence, our definition of twice differentiable functions is the following

$$\mathrm{Diff}_2(f, \mathbf{x}) \iff \exists S, \mathrm{open}(S) \wedge \mathbf{x} \in S \wedge \mathrm{Diff}(f, S) \wedge \left(\forall i, \mathrm{Diff}\left(\frac{\partial f}{\partial x_i}, S\right)\right).$$

We also define twice continuously differentiable functions by adding the continuity condition for second-order partial derivatives at the point $\mathbf{x}$ (the notation is $f \in C^2(\mathbf{x})$). We generalize differentiability properties for arbitrary domains in a natural way.

Continuity of second-order partial derivatives is only required in the proof of equality of mixed partial derivatives

$$f \in C^2(\mathbf{x}) \implies \frac{\partial^2 f}{\partial x_i x_j}(\mathbf{x}) = \frac{\partial^2 f}{\partial x_j x_i}(\mathbf{x}).$$

This is the most difficult theorem in our formalization and the proof is based on the notes [Fel].

Also we show that a convex function attains maximum at the boundary

$$\begin{aligned}\big(\forall i,\ 1 \le i \le n\ \wedge\ i \ne j \implies u_i = z_i\ \wedge\ v_i = x_i\big)\ \wedge\ u_j = x_j\ \wedge\ v_j = z_j\\ \wedge\ \mathrm{Diff}_2(f, [\mathbf{x}, \mathbf{z}])\ \wedge\ \left(\forall \mathbf{y},\ \mathbf{y} \in [\mathbf{x}, \mathbf{z}] \implies 0 \le \frac{\partial^2 f}{\partial x_j \partial x_j}(\mathbf{y})\right)\\ \wedge\ (\forall \mathbf{y},\ \mathbf{y} \in [\mathbf{x}, \mathbf{u}] \implies f(\mathbf{y}) \le h)\ \wedge\ (\forall \mathbf{y},\ \mathbf{y} \in [\mathbf{v}, \mathbf{z}] \implies f(\mathbf{y}) \le h)\\ \implies (\forall \mathbf{y},\ \mathbf{y} \in [\mathbf{x}, \mathbf{z}] \implies f(\mathbf{y}) \le h).\end{aligned}$$

Formalization of Taylor approximations starts with the following theorem about Taylor bounds of a univariate function

$$f \in C^2([0,1])\ \wedge\ (\forall t,\ t \in [0,1] \implies |f''(t)| \le h) \implies \big|f(1) - \big(f(0) + f'(0)\big)\big| \le \frac{h}{2}.$$

This theorem is sufficient for proving bounds of multivariate function if we consider the composition $f \circ (\lambda t.\ \mathbf{x} + t\mathbf{v})$. The second-order derivative of this composition can be computed with the following theorem

$$\mathrm{Diff}_2(f, \mathbf{x} + t\mathbf{v}) \implies \big(f \circ (\lambda t.\ \mathbf{x} + t\mathbf{v})\big)''(t) = \sum_{i,j=1}^{n} v_i v_j \frac{\partial^2 f}{\partial x_j \partial x_i}(\mathbf{x} + t\mathbf{v}).$$

This lemma leads to the following definition of the error term of a Taylor approximation in the form of a predicate

$$\mathrm{TE}(f, \mathbf{a}, \mathbf{b}, \mathbf{w}, e) \iff \left(\forall \mathbf{x},\ \mathbf{x} \in [\mathbf{a}, \mathbf{b}] \implies \sum_{i=1}^{n}\Big(w_i \sum_{j=1}^{n} w_j \left|\frac{\partial^2 f}{\partial x_j \partial x_i}(\mathbf{x})\right| \le e\Big)\right).$$

(The corresponding HOL Light constant is called `m_taylor_error`.) This definition is general. An equivalent definition which involves less computations is derived for twice continuously differentiable functions

$$f \in C^2([\mathbf{a},\mathbf{b}]) \implies \Big(\mathrm{TE}(f,\mathbf{a},\mathbf{b},\mathbf{w},e) \iff \Big(\forall \mathbf{x},\ \mathbf{x} \in [\mathbf{a},\mathbf{b}] \implies \sum_{i=1}^{n} w_i\Big(w_i \left|\frac{\partial^2 f}{\partial x_i \partial x_i}(\mathbf{x})\right| + 2\sum_{j=1}^{i-1} w_j \left|\frac{\partial^2 f}{\partial x_j \partial x_i}(\mathbf{x})\right|\Big) \le e\Big)\Big).$$

To clarify the role of $\mathbf{w}$, we define the following predicate for a rectangular domain $[\mathbf{x},\mathbf{z}]$

$$\mathrm{CD}(\mathbf{x},\mathbf{z},\mathbf{y},\mathbf{w}) \iff (\forall i,\ 1 \le i \le n \implies x_i \le y_i \le z_i \ \wedge\ \max\{y_i - x_i, z_i - y_i\} \le w_i)\,.$$

(CD means "cell domain" and the corresponding HOL Light constant is `m_cell_domain`.) Here, we require that $\mathbf{y} \in [\mathbf{x},\mathbf{z}]$ and that $|y_i - x_i| \le w_i$ and $|z_i - y_i| \le w_i$. Usually, we always have $\mathbf{y} = (\mathbf{x}+\mathbf{z})/2$, i.e., $\mathbf{y}$ is the center of the domain. Due to rounding errors of floating-point arithmetic, $\mathbf{y}$ is not exactly the center, but the definition above still works for carefully chosen values of $\mathbf{w}$.

With all these definitions, we prove theorems for bounds of a given function on a rectangular domain

$$\begin{aligned}\mathrm{CD}(\mathbf{x},\mathbf{z},\mathbf{y},\mathbf{w}) \ \wedge\ \mathrm{Diff}_2(f,[\mathbf{x},\mathbf{z}]) \ \wedge\ \mathrm{TE}(f,\mathbf{x},\mathbf{z},\mathbf{w},e)&\\ \wedge\ f(\mathbf{y}) \le h \ \wedge\ h + \sum_{i=1}^{n}\left(w_i + \left|\frac{\partial f}{\partial x_i}(\mathbf{y})\right|\right) + \frac{e}{2} \le b&\\ \implies (\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{x},\mathbf{z}] \implies f(\mathbf{p}) \le b)\,.&\end{aligned}$$

We also prove the corresponding theorem for the lower bound. Partial derivatives also can be bounded

$$\begin{aligned}\mathrm{CD}(\mathbf{x},\mathbf{z},\mathbf{y},\mathbf{w}) \ \wedge\ \mathrm{Diff}_2(f,[\mathbf{x},\mathbf{z}]) \ \wedge\ \mathrm{TE}_i(f,\mathbf{x},\mathbf{z},\mathbf{w},e) \ \wedge\ a \le \frac{\partial f}{\partial x_i}(\mathbf{y}) \le b&\\ \wedge\ l \le l - e \ \wedge\ b + e \le h \implies \left(\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{x},\mathbf{z}] \implies l \le \frac{\partial f}{\partial x_i}(\mathbf{p}) \le h\right).&\end{aligned}$$

The predicate TE_i estimates the error term for the i-th partial derivative

$$\mathrm{TE}_i(f,\mathbf{x},\mathbf{z},\mathbf{w},e) \iff \left(\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{x},\mathbf{z}] \implies \sum_{j=1}^{n} w_j \left|\frac{\partial^2 f}{\partial x_j \partial x_i}(\mathbf{p})\right| \le e\right).$$

Definitions which packs all data (approximations of first-order partial derivatives at a point, second-order partial derivatives at an interval) are the following. The predicate LA ("linear approximation", `m_lin_approx` in HOL Light) gives approximations of a function and its first partial derivatives at a point

$$\mathrm{LA}(f, \mathbf{y}, f^{lo}, f^{hi}, [(f_1^{lo}, f_1^{hi}); \ldots; (f_n^{lo}, f_n^{hi})]) \iff \\ \Big(\mathrm{Diff}(f, \mathbf{y}) \ \wedge\ f^{lo} \le f(\mathbf{y}) \le f^{hi} \ \wedge\ \big(\forall i,\ 1 \le i \le n \implies f_i^{lo} \le \frac{\partial f}{\partial x_i}(\mathbf{y}) \le f_i^{hi}\big)\Big).$$

Then we define the predicate B_2 (`second_bounded` in HOL Light) for bounds of second-order partial derivatives

$$\mathrm{B}_2\big(f, \mathbf{x}, \mathbf{z}, [[f_{1,1}^{lo}, f_{1,1}^{hi}]; [f_{2,1}^{lo}, f_{2,1}^{hi}; f_{2,2}^{lo}, f_{2,2}^{hi}]; \ldots; [f_{n,1}^{lo}, f_{n,1}^{hi}; \ldots; f_{n,n}^{lo}, f_{n,n}^{hi}]]\big) \iff \\ \Big(\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{x}, \mathbf{z}] \implies \big(\forall i\ j,\ 1 \le i \le n \ \wedge\ j \le i \implies f_{i,j}^{lo} \le \frac{\partial^2 f}{\partial x_j \partial x_i}(\mathbf{p}) \le f_{i,j}^{hi}\big)\Big).$$

The last argument of this predicate is a list of lists of real numbers. Finally, we formally define Taylor interval approximations (`m_taylor_interval` in HOL Light)

$$\mathrm{TI}(f, \mathbf{x}, \mathbf{z}, \mathbf{y}, \mathbf{w}, f^{lo}, f^{hi}, d_{list}, dd_{list}) \iff \\ \mathrm{CD}(\mathbf{x}, \mathbf{z}, \mathbf{y}, \mathbf{w}) \ \wedge\ f \in C^2([\mathbf{x}, \mathbf{z}]) \ \wedge\ \mathrm{LA}(f, \mathbf{y}, f^{lo}, f^{hi}, d_{list}) \ \wedge\ \mathrm{B}_2(f, \mathbf{x}, \mathbf{z}, dd_{list}).$$

4.3 FORMAL COMPUTATIONS

In this section, we will demonstrate how theories developed in the previous section can be used for formal computations of interval Taylor approximations. Our main goal is to compute interval bounds for a given function on a fixed rectangular domain.

All theorems have been proved for an arbitrary dimension $\mathbb{R}^n$ (`:real^N` in HOL Light), i.e., the theorems are polymorphic with respect to the dimension parameter n (`:N`). We are not going to explain in details how vectors are formalized in HOL Light (see [Har05]) but the main point is that it is not possible to assert any specific properties of this parameter n (like, $n < 10$ or $n = 3 + 2$). But it is always possible to instantiate n to a particular fixed value (e.g., $n = 3$, $n = 8$). Whenever we need to do some computations involving general theorems, the first thing we do is replace the type parameter n with a fixed value which corresponds to the dimension of a function (or a domain) for which we want to produce some formally computed result. In the text below, we will use a general parameter n, but it is important to remember that every time we work with a fixed value.

4.3.1 Cell domains and Taylor interval bounds

The first step is to construct a theorem $\vdash \mathrm{CD}(\mathbf{x}, \mathbf{z}, \mathbf{y}, \mathbf{w})$ for a given domain $[\mathbf{x}, \mathbf{z}]$. In particular, we want to find $\mathbf{y} \in [\mathbf{x}, \mathbf{z}]$ and $\mathbf{w}$ which estimates the distance between $\mathbf{y}$ and $\mathbf{x}$, $\mathbf{z}$. Formal computations are straightforward. For given $\mathbf{x}$ and $\mathbf{z}$ (represented as lists of floating-point numbers), we compute $\mathbf{y}$ as the result of a componentwise evaluation of $2^{-1} \times (\mathbf{x} + \mathbf{z})$ with formal floating-point operations which return upper approximations (we don't need an equality theorem here). Then we compute $\mathbf{w}_1 = \mathbf{y} - \mathbf{x}$ and $\mathbf{w}_2 = \mathbf{z} - \mathbf{y}$ (again, we use upper approximations). The last step is to get $\mathbf{w} = \max\{\mathbf{w}_1, \mathbf{w}_2\}$ (componentwise maximum) and to prove the following properties

$$\mathbf{x} \le \mathbf{y} \ \wedge\ \mathbf{y} \le \mathbf{z} \ \wedge\ \mathbf{y} - \mathbf{x} \le \mathbf{w}_1 \ \wedge\ \mathbf{z} - \mathbf{y} \le \mathbf{w}_2 \ \wedge\ \mathbf{w}_1 \le \mathbf{w} \ \wedge\ \mathbf{w}_2 \le \mathbf{w}$$

All variables here are lists of floating-point numbers and all operations are componentwise. Also, we need to check that the sizes of all lists are the same and equal to the dimension n ($|\mathbf{x}| = |\mathbf{y}| = |\mathbf{z}| = |\mathbf{w}| = n$). After these properties are proved (with formal procedures), we construct

$$\vdash \mathrm{CD}\big(\mathrm{vector}(\mathbf{x}), \mathrm{vector}(\mathbf{z}), \mathrm{vector}(\mathbf{y}), \mathrm{vector}(\mathbf{w})\big).$$

Here, $\mathrm{vector}(\mathbf{x})$ is a function which converts a list into a vector (the corresponding HOL Light function is `vector:(real)list->real^N`).

Before we describe how Taylor intervals are computed, let's look at formal computations of bounds and errors for a function with an existing formal Taylor interval. First of all, we translate lemmas about bounds and errors into a more computation-friendly form.

$$\begin{aligned}\mathrm{TI}(f,\mathbf{x},\mathbf{z},\mathbf{y},\mathbf{w},&f^{lo},f^{hi},d^{list},dd^{list})\ \wedge\ \mathrm{TE}(f,\mathbf{x},\mathbf{z},\mathbf{w},e)\ \wedge\ \sum_{i=1}^{n} w_i\left|d_i^{list}\right| \le b\\ &\wedge\ b+2^{-1}e\le a\ \wedge\ l\le f^{lo}-a\ \wedge\ f^{hi}+a\le h\\ &\Longrightarrow\ \big(\forall\mathbf{p},\ \mathbf{p}\in[\mathbf{x},\mathbf{z}]\ \Longrightarrow\ f(\mathbf{p})\in[l,h]\big).\end{aligned}$$

Recall that each formal Taylor interval TI has information about the domain $[\mathbf{x},\mathbf{z}]$, a point inside this domain $\mathbf{y}\in[\mathbf{x},\mathbf{z}]$, the width of the domain in the sense $\mathbf{w}\ge\max\{\mathbf{y}-\mathbf{x},\mathbf{z}-\mathbf{y}\}$, interval bounds of the function f^{lo},f^{hi} and its first-order partial derivatives d^{list} (a list of pairs of real numbers) at $\mathbf{y}$, and interval bounds of all second-order partial derivatives dd^{list} (a list of lists of pairs of real numbers).

We prepare instances of this theorem for all possible dimensions (from 1 to 8 in the current implementation). Variables which contain lists are expanded (i.e., if the dimension is 2, then $dd^{list}=[[dd_{1,1}];[dd_{2,1};dd_{2,2}]]$ for variables $dd_{1,1},dd_{2,1},dd_{2,2}$ which are pairs of real (floating-point) numbers), the definition of TE and sums are also expanded, and trivial conditions are automatically proved. (Trivial conditions include assumptions on lengths of lists dd^{list} and d^{list}; these conditions are not shown in the statement of the theorem above.) An example of a prepared theorem for $n=2$ is given below

$$\begin{aligned}&\mathrm{TI}(f,\mathbf{x},\mathbf{z},\mathbf{y},\mathbf{w},f^{lo},f^{hi},[d_1],[[dd_{1,1}];[dd_{2,1};dd_{2,2}]])\\ &\qquad\wedge\ w_1|d_1|+w_2|d_2|\le b\ \wedge\ w_1(w_1|dd_{1,1}|)+w_2(w_2|dd_{2,2}|+2w_1|dd_{2,1}|)\le e\\ &\qquad\wedge\ b+2^{-1}e\le a\ \wedge\ l\le f^{lo}-a\ \wedge\ f^{hi}+a\le h\\ &\qquad\Longrightarrow\ \big(\forall\mathbf{p},\ \mathbf{p}\in[\mathbf{x},\mathbf{z}]\ \Longrightarrow\ f(\mathbf{p})\in[l,h]\big).\end{aligned}$$

It is not necessary to prepare theorems before doing formal computations, but we save time on proving results about lengths of lists and on instantiation of the dimension parameter. The only drawback is that we need to instantiate more variables (e.g., instead of one variable d^{list} we need to instantiate n variables $d_1,\dots,d_n$). Nevertheless the prepared theorems work faster than unprepared ones: the initial implementation was without prepared theorems and it was about 10% slower.

The algorithm for computing bounds with a Taylor interval is straightforward: given a Taylor interval, we compute all quantities in the main (prepared) theorem and then instantiate this theorem with variables taken from the given Taylor interval. Almost all computations are done without interval operations: everywhere we have inequalities, so we may use the corresponding floating-point operations (here, we need only operations which yield upper bounds with one exception for the computation of $l \le f^{lo} - a$). Intervals are only used for computing upper bounds of the corresponding absolute values. These interval operation require one floating-point negation and comparison since $|(l,h)| = \max\{-l, h\}$. The main property of this operation is

$$x \in [l,h] \implies |x| \le |(l,h)|.$$

There are variations of the procedure described above which compute only upper or lower bounds for a given Taylor interval. Similar procedures are defined for formal computations of bounds of partial derivatives. There is one trick: the error term for the i-th partial derivative looks like

$$\sum_{j=1}^{n} w_j \left| \frac{\partial^2 f}{\partial x_j \partial x_i} \right|.$$

For fixed i, the list dd^{list} contains bounds of second-order partial derivatives from 1 to i only. So there is no direct way to encode the error term with one summation operation. The solution of this problem is to use a special theorem where an additional list t^{list} of second-order partial derivatives is given for the error computation. The theorem is the following

$$\begin{aligned}
1 \le i \le n \ \wedge\ & \mathrm{TI}(f, \mathbf{x}, \mathbf{z}, \mathbf{y}, \mathbf{w}, f^{lo}, f^{hi}, d^{list}, dd^{list}) \\
& \wedge \Big(\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{x}, \mathbf{z}] \implies \big(\forall j,\ (j \le i \implies \frac{\partial^2 f}{\partial x_j \partial x_i}(\mathbf{p}) \in [t_j^{lo}, t_j^{hi}]) \\
& \qquad\qquad\qquad\qquad \wedge\ (i < j \implies \frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{p}) \in [t_j^{lo}, t_j^{hi}])\big)\Big) \\
& \wedge \sum_{j=1}^{n} w_j |t_j| \le e \ \wedge\ d_i^{hi} + e \le h \ \wedge\ l \le d_i^{lo} - e \\
& \implies \big(\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{x}, \mathbf{z}] \implies \frac{\partial f}{\partial x_i}(\mathbf{p}) \in [l, h]\big).
\end{aligned}$$

We prepare instances of this theorem for all possible values of the dimension parameter n and for all i. For instance, the prepared theorem for $n = 3$ and $i = 2$ is

$$\begin{aligned}\mathrm{TI}(f, \mathbf{x}, \mathbf{z}, \mathbf{y}, \mathbf{w}, f^{lo}, f^{hi}, [d_1; (d_2^{lo}, d_2^{hi}); d_3], [[dd_{1,1}]; [dd_{2,1}; dd_{2,2}]; [dd_{3,1}; dd_{3,2}; dd_{3,3}]]) \\ \wedge\ w_1|dd_{2,1}| + w_2|dd_{2,2}| + w_3|dd_{3,2}| \le e\ \wedge\ l \le d_2^{lo} - e\ \wedge\ d_2^{hi} + e \le h \\ \implies \Big(\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{x}, \mathbf{z}] \implies \frac{\partial f}{\partial x_2}(\mathbf{p}) \in [l, h]\Big).\end{aligned}$$

As we see, the temporary list t^{list} is completely eliminated and the corresponding entrances of dd^{list} are used.

4.3.2 Formal Taylor intervals

We compute formal Taylor intervals directly for multivariate polynomial functions and then use special procedures for combining existing Taylor intervals and computing approximations for arbitrary functions.

Partial derivatives of polynomials are easy to compute and we use standard HOL Light rewriting and matching functions in order to compute them. Also, we use the standard polynomial simplification function `REAL_POLY_CONV` to simplify results. We only need to compute partial derivatives once, so there is no real reason to optimize this procedure.

When partial derivatives of a polynomial are computed, the next step is to automatically generate special theorems for computing the linear approximation LA of the polynomial at a given point and bounds B_2 for all its second-order partial derivatives on a full domain. Here is an example of these theorems for $f(\mathbf{x}) = x_1 + x_2x_1 - 2$ in 2 dimensions:

$$\begin{aligned}\vdash \mathrm{LA}\big((\lambda \mathbf{x}.\ x_1 + x_2x_1 - 2)(\mathrm{vector}([y_1; y_2]), (f^{lo}, f^{hi}), [d_1; d_2]\big) \\ \iff y_1 + y_2y_1 - 2 \in [f^{lo}, f^{hi}]\ \wedge\ y_2 + 1 \in [d_1^{lo}, d_1^{hi}]\ \wedge\ y_1 \in [d_2^{lo}, d_2^{hi}],\end{aligned}$$

$$\begin{aligned}\vdash \mathrm{B}_2\big((\lambda \mathbf{x}.\ x_1 + x_2x_1 - 2), \mathbf{x}, \mathbf{z}, [[dd_{1,1}]; [dd_{1,2}; dd_{2,2}]]\big) \\ \iff 0 \in [dd_{1,1}^{lo}, dd_{1,1}^{hi}]\ \wedge\ 1 \in [dd_{1,2}^{lo}, dd_{1,2}^{hi}]\ \wedge\ 0 \in [dd_{2,2}^{lo}, dd_{2,2}^{hi}].\end{aligned}$$

Two special procedures `eval_lin_approx` and `eval_second_bounded` build formal interval functions based on the generated theorems and then compute values of these interval functions for any input domain and instantiate variables in the corresponding theorems. The procedure `eval_m_taylor` has the type `:int->thm->thm->thm->(int->int->thm)`. The second parameter is the theorem which tells that the given polynomial function is twice continuously differentiable everywhere (we don't have any special conditions for polynomials which is convenient here); third and fourth parameters are theorems about linear approximation and bounds of second-order partial derivatives; the first parameter is the precision parameter which is used for approximation of all constant expressions in the formulas for a polynomial and its derivatives (for instance, if $f(x) = 2\pi - x$, then 2π will be immediately approximated with the given precision before any other formal computations). `eval_m_taylor p0 diff2_th lin_th second_th` yields a function with three parameters. This function can compute Taylor interval approximations for any given domain (the last argument must contain a theorem in the form `|- m_cell_domain ...`) and with any precision constants (the first argument is the precision constant for computation of the linear approximation, the second argument is the precision constant for computations of bounds of second-order partial derivatives).

Then we define operations which compute sums, products, and apply elementary functions to existing Taylor intervals. More specifically, given Taylor intervals for functions f and g, then there are operations which compute Taylor intervals of $f+g$, $f-g$, $f\times g$, $-f$, $1/f$, $\sqrt{f}$, $\arccos f$, $\arctan f$. Consider how these operations are implemented using the square root operation as an example.

This operation is based on the following theorem

$$\begin{aligned}
\mathrm{CD}(\mathbf{x},\mathbf{z},\mathbf{y},\mathbf{w})\ \wedge\ &\big(\forall \mathbf{p},\ \mathbf{p}\in[\mathbf{x},\mathbf{z}] \implies f^{lo} \le f(\mathbf{p}) \le f^{hi}\big)\ \wedge\ 0 \le f^{lo}\ \wedge\ f \in C^2([\mathbf{x},\mathbf{z}])\\
&\wedge\ s^{lo} \le \sqrt{f(\mathbf{y})} \le s^{hi}\ \wedge\ \Big(\forall i,\ 1 \le i \le n \implies s_i^{lo} \le \frac{1}{2\sqrt{f(\mathbf{y})}}\frac{\partial f}{\partial x_i}(\mathbf{y}) \le s_i^{hi}\Big)\\
&\wedge\ \Big(\forall \mathbf{p},\ \mathbf{p}\in[\mathbf{x},\mathbf{z}] \implies \big(\forall i\ j,\ 1 \le i \le n\ \wedge\ j \le i\\
&\qquad\qquad \implies s_{i,j}^{lo} \le -\frac{1}{2\sqrt{f(\mathbf{p})}}\Big(\frac{1}{2f(\mathbf{p})}\frac{\partial f}{\partial x_j}(\mathbf{p})\frac{\partial f}{\partial x_i}(\mathbf{p}) + \frac{\partial^2 f}{\partial x_j \partial x_i}(\mathbf{p})\Big) \le s_{i,j}^{hi}\big)\Big)\\
&\implies \mathrm{TI}\big(\sqrt{f},\mathbf{x},\mathbf{z},\mathbf{y},\mathbf{w},s^{lo},s^{hi},[s_1^{lo},s_1^{hi};\ldots;s_n^{lo},s_n^{hi}],\\
&\qquad\qquad [[s_{1,1}^{lo},s_{1,1}^{hi}];\ldots;[s_{n,1}^{lo},s_{n,1}^{hi};\ldots;s_{n,n}^{lo},s_{n,n}^{hi}]]\big).
\end{aligned}$$

As before, we are not using this theorem directly but first prepare different versions of this theorem for fixed dimensions (from 1 to 8). This preparation yields theorems which contain a simple conjunction of required inequalities.

The algorithm for computing Taylor intervals for the square root is straightforward. It takes two precision parameters (for linear computations and for second-order computations) and a theorem containing a Taylor interval approximation for a given function f on some domain. The interesting part is how bounds of second-order partial derivatives of the square root are computed. In order to compute these bounds, it is necessary to know bounds of the function and its first partial derivatives on the full domain. This information is not directly encoded in the input Taylor interval but it can be computed using procedures for bound estimates of a Taylor interval (that's why we have procedures for computing both upper and lower bounds meanwhile we only need upper bounds to prove inequalities $f(\mathbf{x}) < 0$).

The square root operation fails if the estimated interval of $f(\mathbf{x})$ contains non-positive numbers. Operations $+$, $-$, $\times$, and arctan never fail.

Finally, for a given HOL Light term which defines a function, we have a special procedure which constructs the corresponding function for computing formal Taylor interval approximations. The constructed evaluation procedure is quite efficient since it automatically finds all polynomial subterms of the original term and builds the corresponding polynomial Taylor intervals. For non-polynomial operations, the constructed procedure invokes the corresponding formal Taylor interval operations.

4.4 FORMAL VERIFICATION

The formal verification procedure has the following arguments: a procedure for computing formal Taylor intervals, a solution certificate, a domain theorem on which the inequality should be verified. For each solution certificate node, the formal verification algorithm performs the corresponding formal verification step. Here is the algorithm

```
let verify f certificate dom rs =
   match certificate with
```

```
| Result_pass ->
   let taylor_interval = {evaluate f on dom}
     {verify that the upper bound of taylor_interval is less than 0}
| Result_mono (mono_list, c) ->
   let taylor_interval = {evaluate f on dom}
   let partial_bounds = {evaluate bounds of partial derivatives
                          with taylor_interval}
   let new_dom = {restrict dom based on monotonicity data}
   let result0 = verify f c new_dom rs
      {use the monotonicity argument to prove the bound of f on dom}
| Result_glue (i, convex_flag, c1, c2) ->
   let dom1, dom2 =
      if convex_flag then
         {restrict dom along i}
      else
         {split dom along i}
   let r1 = verify f c1 dom1 rs
   let r2 = verify f c2 dom2 rs
   if convex_flag then
      {glue r1 and r2 with the convexity argument}
   else
      {glue r1 and r2}
| Result_ref j ->
   let r = {get the j-th item of the list rs}
     {prove that dom is a subset of the domain in r
      and return the result for dom}
```

The real algorithm has the precision parameter which is passed to the function which evaluates formal Taylor intervals and to the function which splits the domain. We omit this parameter in our description. A detailed explanation of the algorithm is the following. If the current certificate node is `Result_pass`, then the formal Taylor interval is computed for the current domain D (`dom` in the algorithm) and the upper bound of the function is computed on the domain. This upper bound is compared with 0. If it is less than 0, then the theorem $\vdash \forall \mathbf{x}, \mathbf{x} \in D \implies f(\mathbf{x}) < 0$ is generated and returned. If the upper bound is greater or equals to zero, then the algorithm fails with an error message.

The `Result_mono` node is processed in the following way. This node has a list of parameters specifying which partial derivatives are positive or negative on the current domain. The first verification step is to compute the formal Taylor interval for the current domain. Then bounds of specified partial derivatives are verified with this Taylor interval. The next step is to create a new restricted domain. This restriction is done iteratively for all elements in the list of monotonicity parameters. For instance, suppose that we have data that the j-th partial derivative is positive. If the original domain is defined by parameters (x_i, z_i, y_i, w_i) (as always, the domain itself is defined by $[\mathbf{x}, \mathbf{z}]$, $\mathbf{y} \in [\mathbf{x}, \mathbf{z}]$, and $\max\{\mathbf{y} - \mathbf{x}, \mathbf{z} - \mathbf{y}\} \le \mathbf{w}$), then the parameters of the restricted domain will be $x_i^{(j)} = x_i$, $y_i^{(j)} = y_i$, $w_i^{(j)} = w_i$ for $i \neq j$, $x_j^{(j)} = y_j^{(j)} = z_j$, $w_j^{(j)} = 0$, and $\mathbf{z}^{(j)} = \mathbf{z}$. Formally, we have the following theorems which help to verify that the restricted domains are domains and subsets of the original domain:

$$\begin{aligned}
\mathrm{CD}(\mathbf{x}, \mathbf{z}, \mathbf{y}, \mathbf{w}) \ \wedge \ \big(\forall i,\ 1 \le i \le n \ \wedge \ i \neq j \implies u_i = x_i \ \wedge \ y'_i = y_i \ \wedge \ w'_i = w_i\big) \\
\wedge \ u_j = z_j \ \wedge \ y'_j = z_j \ \wedge \ w'_j = 0 \\
\implies \mathrm{CD}(\mathbf{u}, \mathbf{z}, \mathbf{y}', \mathbf{w}') \ \wedge \ [\mathbf{u}, \mathbf{z}] \subset [\mathbf{x}, \mathbf{z}],
\end{aligned}$$

$$\begin{aligned}
\mathrm{CD}(\mathbf{x}, \mathbf{z}, \mathbf{y}, \mathbf{w}) \ \wedge \ \big(\forall i,\ 1 \le i \le n \ \wedge \ i \neq j \implies u_i = z_i \ \wedge \ y'_i = y_i \ \wedge \ w'_i = w_i\big) \\
\wedge \ u_j = x_j \ \wedge \ y'_j = x_j \ \wedge \ w'_j = 0 \\
\implies \mathrm{CD}(\mathbf{x}, \mathbf{u}, \mathbf{y}', \mathbf{w}') \ \wedge \ [\mathbf{x}, \mathbf{u}] \subset [\mathbf{x}, \mathbf{z}].
\end{aligned}$$

The first theorem is for increasing functions (positive partial derivatives), the second theorem is for decreasing functions. We create a table for instances of these theorems for all possible dimensions and indices of partial derivatives. These special versions of theorems are simpler than general theorems because all vector variables are replaced with corresponding component variables. For instance, a two dimensional version of the first theorem for $j = 2$ is the following

$$\begin{aligned}
&\mathrm{CD}\big(\langle x_1, x_2\rangle, \langle z_1, z_2\rangle, \langle y_1, y_2\rangle, \langle w_1, w_2\rangle\big) \\
&\qquad \implies \mathrm{CD}\big(\langle x_1, z_2\rangle, \langle z_1, z_2\rangle, \langle y_1, z_2\rangle, \langle w_1, 0\rangle\big) \ \wedge\ [\langle x_1, z_2\rangle, \langle z_1, z_2\rangle] \subset [\langle x_1, x_2\rangle, \langle z_1, z_2\rangle].
\end{aligned}$$

After the restricted domain is computed and the corresponding theorems are verified, we verify the inequality on the restricted domain with the certificate given as the second parameter of `Result_mono`. The result on the original domain is verified with the following theorem (this is the increasing function case, the decreasing function case is analogous)

$$\begin{aligned}
[\mathbf{x}', \mathbf{z}'] \subset [\mathbf{x}, \mathbf{z}] \ \wedge\ & f \in C^2([\mathbf{x}, \mathbf{z}]) \\
&\wedge\ \big(\forall i,\ 1 \le i \le n \ \wedge\ i \ne j \implies u_i = x_i\big) \ \wedge\ u_j = z_j \\
&\wedge\ 0 \le l \ \wedge\ \big(\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{x}, \mathbf{z}] \implies l \le \frac{\partial f}{\partial x_j}(\mathbf{p})\big) \\
&\wedge\ \big(\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{u}, \mathbf{z}] \implies f(\mathbf{p}) < 0\big) \\
&\implies \big(\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{x}', \mathbf{z}'] \implies f(\mathbf{p}) < 0\big).
\end{aligned}$$

Again, simplified instances of this theorem (which contain components of domain vectors explicitly) are prepared for all dimensions and partial derivative indices and saved in tables. The fact that f is twice continuously differentiable on the domain follows from the computed formal Taylor interval (the definition of a formal Taylor interval includes this condition).

The node `Result_glue` contains two parameters and two certificates. The first parameter j specifies the coordinate along which the domain should be sliced. The second parameter is a boolean flag which indicates if the convexity argument may be used to reduce the dimension of new subdomains. If the convexity flag is false, then two new subdomains are created by splitting the current domain along the given coordinate. The parameters $\mathbf{x}^{(1),(2)}$ and $\mathbf{z}^{(1),(2)}$ of new subdomains are computed informally as $\mathbf{x}^{(1)} = \mathbf{x}$, $z_i^{(1)} = z_i$ if $i \ne j$ and $z_j^{(1)} = y_j$; $\mathbf{z}^{(2)} = \mathbf{z}$, $x_i^{(2)} = x_i$ if $i \ne j$ and $x_j^{(2)} = y_j$. Two other parameters and the corresponding domain theorems are obtained as results of the standard formal procedure for constructing `m_cell_domain`. If the convexity flag is true, then new subdomains are computed as left and right restrictions along the j-th coordinate of the original domain.

In the next step, the inequality is verified on two subdomains. Each verification returns a theorem in the form $\vdash \forall \mathbf{x},\ \mathbf{x} \in D^{1,2} \implies f(\mathbf{x}) < 0$. It is left to glue these two results together. In the case of false convexity flag, gluing is done with the following theorem

$$\begin{aligned}\big(\forall i,\ 1 \le i \le n \ \wedge\ i \ne j \implies u_i = x_i \ \wedge\ v_i = z_i\big) \ \wedge\ v_j = u_j \\ \wedge\ \big(\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{x}, \mathbf{v}] \implies f(\mathbf{p}) < 0\big) \ \wedge\ \big(\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{u}, \mathbf{z}] \implies f(\mathbf{p}) < 0\big) \\ \implies \big(\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{x}, \mathbf{z}] \implies f(\mathbf{p}) < 0\big).\end{aligned}$$

The convexity case is a little more difficult. We have the theorem

$$\begin{aligned}f \in C^2([\mathbf{x}, \mathbf{z}]) \ \wedge\ \big(\forall i,\ 1 \le i \le n \ \wedge\ i \ne j \implies u_i = z_i \ \wedge\ v_i = x_i\big) \ \wedge\ u_j = x_j \ \wedge\ v_j = z_j \\ \wedge\ \big(\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{x}, \mathbf{z}] \implies 0 \le \frac{\partial^2 f}{\partial x_j \partial x_j}(\mathbf{p})\big) \\ \wedge\ \big(\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{x}, \mathbf{u}] \implies f(\mathbf{p}) < 0\big) \ \wedge\ \big(\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{v}, \mathbf{z}] \implies f(\mathbf{p}) < 0\big) \\ \implies \big(\forall \mathbf{p},\ \mathbf{p} \in [\mathbf{x}, \mathbf{z}] \implies f(\mathbf{p}) < 0\big).\end{aligned}$$

To apply this theorem, it is necessary to check that the corresponding second partial derivative is positive. This verification is done by a direct interval evaluation of the second partial derivative. As before, simplified versions of gluing theorems are prepared for all dimensions and indices.

To explain how `Result_ref` works, we need to recall that some solution certificates may be transformed into a list of certificates where each element of the list contains a solution certificate and information about a subdomain for which this certificate works. Certificates in the list may refer to each other. Moreover, the i-th certificate may refer only to j-th certificate if $j < i$. That means that the first element of the list contains no references and may be completely formally verified. The next element in the list may refer to the first proved result, and so on. We need to call the formal verification procedure for each element of the list and save all results in another list. The last result will always contain the theorem for the full domain. When we call the verification procedure for the j-th certificate, we pass all previous results as the last argument of the procedure, `rs`. If the certificate contains `Result_ref k` then $k < j$ and we get the referred result from the list of proved results. In general, `Result_ref` must be verified on a smaller domain than the referred result. The following trivial theorem resolves this issue

$$A \subset B \wedge (\forall \mathbf{x},\ \mathbf{x} \in B \implies f(\mathbf{x}) < 0) \implies (\forall \mathbf{x},\ \mathbf{x} \in A \implies f(\mathbf{x}) < 0).$$

We only need to prove that one domain is a subset of another domain. All our domains are rectangles, so the subset relation is proved by verifying several simple inequalities.

Here is the algorithm which verifies an inequality based on a list of certificates

```
let verify_list f certificate_list dom0 =
  let verify_all list results =
    match list with
    | [] -> last results
    | (path, certificate) :: tail ->
      let dom = {find domain defined by path in dom0}
      let th = verify f certificate dom results
      let next_results = {attach th to the end of results}
        verify_all tail next_results
    verify_all certificate_list []
```

Consider an example. Let $f(x) = x - 2$ and we want to prove $f(x) < 0$ for $x \in [-1, 1]$. Suppose that we have the following solution certificate

```
Result_glue {1, false,
   Result_pass_mono {[1, incr]},
   Result_mono {[1, incr],
      Result_pass
   }
}
```

This certificate tells that the inequality may be verified by first splitting the domain into two subdomains along the first (and the only) variable; then the left branch follows from some other verification by monotonicity (`Result_pass_mono`); the right branch follows by the monotonicity argument and by a direct verification. This certificate cannot be used directly for a formal verification since we don't know how the left branch is proved. The first step is to transform this certificate into a list of certificate such that each certificate can be verified on subdomains specified by the corresponding paths. We get the following list of certificates

```
[
  ["r", 1], Result_mono {[1], Result_pass};
  ["l", 1], Result_mono {[1], Result_ref {0}};
  [], Result_glue {1, false, Result_ref {1}, Result_ref {0}}
]
```

The first element corresponds to the right branch of the original `Result_glue` (hence, the path is `["r", 1]` which means subdivision along the first variable and taking the right subdomain). A formal verification of the first certificate yields $\vdash x \in [0,1] \implies f(x) < 0$. The second result is the transformed left branch of the original certificate. This transformed result explicitly refers to the first proved result (`Result_ref {0}`). Now it can be verified. Indeed, `Result_ref {0}` yields $\vdash x \in [0,0] \implies f(x) < 0$ (since $[0,0] \subset [0,1]$ and we have the theorem for $[0,1]$ which we use in the reference). Then the monotonicity argument

$$(\forall x,\ x \in [-1,0] \implies 0 \le f'(x)) \wedge (\forall x, x \in [0,0] \implies f(x) < 0) \\ \implies (\forall x, x \in [-1,0] \implies f(x) < 0)$$

yields $\vdash x \in [-1,0] \implies f(x) < 0$. The last entry of the list refers to two proved results and glues them together in the right order.

4.5 OPTIMIZATIONS AND FUTURE WORK

4.5.1 Implemented optimization techniques

There are several optimization techniques for formal verification of nonlinear inequalities. One of the basic ideas of optimization techniques is to compute extra information for solution certificates which helps to increase the performance of formal verification procedures.

The first optimization technique is to try out direct interval evaluations without Taylor approximations. If a direct interval evaluation yields a desired result (verification of an inequality on a domain or verification of a monotonicity property), then a special flag is added to the corresponding certificate node. This flag indicates that it is not necessary to compute full formal Taylor interval and it is enough to evaluate the function directly with interval arithmetic (which is faster). These flags are added to `Result_pass` and `Result_mono` nodes.

Cached formal arithmetic significantly increases the performance of verification procedures (see test result in the next section). It can be explained by the fact that formal evaluations of a function on subdomains obtained by splitting another domain in half share the same values for many variables.

An important optimization procedure is to find the best (minimal) precision which is sufficient for verifying an inequality on each subdomain. We have a special informal implementation of all arithmetic, Taylor interval evaluation, and verification functions which compute results in the same way as the corresponding formal functions. This informal implementation is much simpler (because it does not prove anything) and faster (since it does not prove anything and all basic arithmetic is done by native machine arithmetic). For a given solution certificate, we run a modified informal verification procedure which tests different precision parameter values for each certificate node. It finds out the smallest value of the precision parameter for each certificate node such that the verification result is correct. Then a modified solution certificate is created where each node contains information about the best precision parameter. A special version of the formal verification procedure accepts this new certificate and verifies the inequality with computed precision parameters. This adaptive precision technique increases the performance of formal arithmetic computations.

4.5.2 Future work

There are some optimization ideas which are not implemented yet. The first idea is to stop computations of bounds of second-order partial derivatives for Taylor intervals at some point and reuse bounds computed for larger domains. The error term in Taylor approximation depends quadratically on the size of a domain. When domains are sufficiently small, good approximations of bounds of second-order partial derivatives are not very important. This strategy could save quite a lot of verification time since formal evaluation of second-order partial derivative bounds is expensive for many functions. Some tests were performed to see how well this approach works with existing solutions certificates. It was shown that about 20% of evaluations of second-order partial derivative bounds may be eliminated in average. If this strategy is applied to the certificate search procedure, these results could be even better.

Another unimplemented optimization is verification of sets of similar inequalities on the same domain. The idea is to reuse results of formal computations as much as possible for inequalities which have a similar structure and which are verified on the same domains. The basic strategy is to find a subdivision of the domain into subdomains such that each inequality in the set can be completely verified on each subdomain. If inequalities in the set share a lot of similar computations, then the verification of all inequalities in the set could be almost as fast as the verification of the most difficult inequality in the set. This approach should work well for Flyspeck inequalities where many inequalities share the same sub-expressions and domains.

An important unimplemented feature is verification of disjunctions of inequalities. That is, we want to verify inequalities in the form

$$\forall \mathbf{x} \in D \implies f_1(\mathbf{x}) < 0 \ \lor \ f_2(\mathbf{x}) < 0 \ \lor \ \ldots \ \lor \ f_k(\mathbf{x}) < 0.$$

This form is equivalent to an inequality on a non-rectangular domain since

$$(P(\mathbf{x}) \implies f(\mathbf{x}) < 0 \ \lor \ g(\mathbf{x}) < 0) \iff (P(\mathbf{x}) \ \land \ 0 \le g(\mathbf{x}) \implies f(\mathbf{x}) < 0).$$

Many Flyspeck inequalities are in this form. A formal verification of these inequalities is simple. It is enough to add indices of functions for which the inequality is satisfied to the corresponding nodes of solution certificates. Then it will be only necessary to modify the formal gluing procedure. It should be able to combine inequalities for different functions with disjunctions.

4.6 OVERVIEW OF THE FORMAL VERIFICATION TOOL

A user manual which contains information about the tool and installation instructions is available at [Sol12b]. Here, we briefly describe how the tool can be used.

Suppose we want to verify a polynomial inequality

$$-\frac{1}{\sqrt{3}} \le x \le \sqrt{2},\ -\sqrt{\pi} \le y \le 1 \implies x^2y - xy^4 + y^6 + x^4 - 7 > -7.17995.$$

The following HOL Light script solves this problem

```
needs "verifier/m_verifier_main.hl";;
open M_verifier_main;;


let ineq =
  `-- &1 / sqrt(&3) <= x /\ x <= sqrt(&2) /\
   -- sqrt(pi) <= y /\ y <= &1
   ==> x pow 2 * y - x * y pow 4 + y pow 6 - &7 + x pow 4 > -- #7.17995`;;


let th, stats = verify_ineq default_params 5 ineq;;
```

First two lines of the script load the verification tool. The main verification function is called `verify_ineq`. It takes 3 arguments. The first argument contains verification options. In most cases, it is enough to provide default options `default_params`. The second parameter specifies the precision of formal floating-point operations. The third parameter is the inequality itself given as a HOL Light term. The format of this term is simple: it is an implication with bounds of variables in the antecedent and an inequality in the consequent. The bounds of all variables should be in the form *a constant expression* $\le x$ or $x \le$ *a constant expression*. For each variable, upper and lower bounds must be given. The inequality must be a strict inequality ($<$ or $>$). The inequality may include `sqrt` ($\surd$), `atn` (arctan), and `acs` (arccos) functions. The constant `pi` (π) is also allowed.

The verification function returns a HOL Light theorem and a record with some verification information which includes verification time.

4.7 TESTS

This section contains performance test results for several polynomial and non-polynomial inequalities. All tests were performed on Intel Core i5, 2.67GHz running Ubuntu 9.10 inside Virtual Box 4.2.0 on a Windows 7 host; the Ocaml version was 3.09.3; the base of arithmetic was 200; the caching was turned on (for most tests).

4.7.1 Polynomial inequalities

Here is a list of test polynomial inequalities taken from [MN12].

- **schwefel**

$$\begin{aligned}&\langle x_1, x_2, x_3\rangle \in [\langle -10, -10, -10\rangle, \langle 10, 10, 10\rangle] \\ &\qquad \implies -5.8806 \times 10^{-10} < (x_1 - x_2^2)^2 + (x_2 - 1)^2 + (x_1 - x_3^2)^2 + (x_3 - 1)^2.\end{aligned}$$

- **rd**

$$\begin{aligned}&(x_1, x_2, x_3) \in [\langle -5, -5, -5\rangle, \langle 5, 5, 5\rangle] \\ &\qquad \implies -36.7126907 < -x_1 + 2x_2 - x_3 - 0.835634534\, x_2(1 + x_2).\end{aligned}$$

- **caprasse**

$$\begin{aligned}&\langle x_1, x_2, x_3, x_4\rangle \in [\langle -0.5, -0.5, -0.5, -0.5\rangle, \langle 0.5, 0.5, 0.5, 0.5\rangle] \\ &\qquad \implies -3.1801 < -x_1x_3^3 + 4x_2x_3^2x_4 + 4x_1x_3x_4^2 + 2x_2x_4^3 \\ &\qquad\qquad + 4x_1x_3 + 4x_3^2 - 10x_2x_4 - 10x_4^2 + 2.\end{aligned}$$

- **lv**

$$\begin{aligned}&\langle x_1, x_2, x_3, x_4\rangle \in [\langle -2, -2, -2, -2\rangle, \langle 2, 2, 2, 2\rangle] \\ &\qquad \implies -20.801 < x_1x_2^2 + x_1x_3^2 + x_1x_4^2 - 1.1x_1 + 1.\end{aligned}$$

Table 7: Polynomial inequalities

Inequality ID	# variables	total time (s)	formal (s)	formal (no caching) (s)
schwefel	3	26.329	19.145	55.987
rd	3	1.593	0.017	0.023
caprasse	4	8.057	1.286	3.762
lv	4	1.875	0.030	0.060
magnetism	7	7.007	1.347	4.916
heart	8	17.298	1.277	3.101

- **magnetism**

$$\begin{aligned}\langle x_1, x_2, x_3, x_4, x_5, x_6, x_7\rangle &\in [\langle -1,-1,-1,-1,-1,-1,-1\rangle, \langle 1,1,1,1,1,1,1\rangle] \\ &\implies -0.25001 < x_1^2 + 2x_2^2 + 2x_3^2 + 2x_4^2 + 2x_5^2 + 2x_6^2 + 2x_7^2 - x_1.\end{aligned}$$

- **heart**

$$\begin{aligned}\langle x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\rangle \in [&\langle -0.1, 0.4, -0.7, -0.7, 0.1, -0.1, -0.3, -1.1\rangle, \\ &\langle 0.4, 1, -0.4, 0.4, 0.2, 0.2, 1.1, -0.3\rangle] \\ &\implies -1.7435 < -x_1x_6^3 + 3x_1x_6x_7^2 - x_3x_7^3 + 3x_3x_7x_6^2 - x_2x_5^3 \\ &\qquad + 3x_2x_5x_8^2 - x_4x_8^3 + 3x_4x_8x_5^2 - 0.9563453.\end{aligned}$$

Performance test results are given in Table 7. All tests were performed with the precision parameter $p = 5$. The column *total time* contains total verification time, the column *formal* contains time of the formal verification only. The formal verification excludes all preliminary processes: computations of partial derivatives, search of solution certificates, adaptive precision search procedures. The column *formal (no caching)* shows formal verification time when caching of formal arithmetic operations was turned off.

4.7.2 Flyspeck inequalities

The Flyspeck project contains 985 nonlinear inequalities. The informal verification program written in C++ by T. Hales can verify all these inequalities in about 10 hours. Most inequalities (683) can be informally verified in less than 10 seconds. Almost all inequalities (911) can be informally verified in less than 100 seconds. There are 3 difficult inequalities (their labels are `9075398267`, `2065952723 A1`, and `5556646409`) which require 1015, 3869, and 4445 seconds for the informal verification.

We tested our formal verification procedure on several simple Flyspeck inequalities. Some of these inequalities are listed below. Table 8 contains performance test results for these inequalities. The column *total time* contains total formal verification time, the column *formal* contains time of the formal verification only (excluding all preliminary processes), the column *informal* contains informal verification time by the C++ program.

$$\begin{aligned}
\Delta(x_1,\ldots,x_6) &= x_1x_4(-x_1+x_2+x_3-x_4+x_5+x_6)\\
&\quad +x_2x_5(x_1-x_2+x_3+x_4-x_5+x_6)\\
&\quad +x_3x_6(x_1+x_2-x_3+x_4+x_5-x_6)\\
&\quad -x_2x_3x_4-x_1x_3x_5-x_1x_2x_6-x_4x_5x_6,\\
\Delta_4 &= \frac{\partial\Delta}{\partial x_4},
\end{aligned}$$

$$\begin{aligned}
\mathrm{dih}_x(x_1,\ldots,x_6) &= \frac{\pi}{2}-\arctan\left(\frac{-\Delta_4(x_1,\ldots,x_6)}{\sqrt{4x_1\Delta(x_1,\ldots,x_6)}}\right),\\
\mathrm{dih}_y(y_1,\ldots,y_6) &= \mathrm{dih}_x(y_1^2,\ldots,y_6^2).
\end{aligned}$$

- **4717061266**

$$4\le x_i\le 6.3504 \implies \Delta(x_1,x_2,x_3,x_4,x_5,x_6)>0.$$

- **7067938795**

$$4 \le x_{1,2,3} \le 6.3504,\ x_4 = 4,\ 3.01^2 \le x_{5,6} \le 3.24^2$$
$$\implies \operatorname{dih}_x(x_1, \ldots, x_6) - \pi/2 + 0.46 < 0.$$

- **3318775219**

$$2 \le y_i \le 2.52 \implies 0 < \operatorname{dih}_y(y_1, \ldots, y_6) - 1.629 + 0.414(y_2 + y_3 + y_5 + y_6 - 8.0)$$
$$- 0.763(y_4 - 2.52) - 0.315(y_1 - 2.0).$$

We also found formal verification time of all Flyspeck inequalities which can be verified in less than one second and which do not contain disjunctions of inequalities. Table 9 summarizes test results.

Test results show that our formal verification procedure is about 2000–4000 times slower than the informal verification program.

Table 8: Flyspeck inequalities

Inequality ID	precision	total time (s)	formal (s)	informal (s)
2485876245a	4	5.530	0.058	0
4559601669b	4	4.679	0.048	0
4717061266	4	27.1	0.250	0
5512912661	4	8.860	0.086	0.002
6096597438a	4	0.071	0.071	0
6843920790	4	2.824	0.076	0.002
SDCCMGA b	4	9.012	0.949	0.006
7067938795	4	431	387	0.070
5490182221	4	1726	1533	0.375
3318775219	4	17091	15226	8.000

Table 9: Flyspeck inequalities which can be informally verified in 1 second

time interval (ms)	# inequalities	total time (s)	formal (s)	informal (s)
0	57	423	2.159	0
1–100	35	5546	3854	1.134
101–500	11	12098	10451	3.944
501–700	14	32065	28705	8.423
701–1000	9	19040	16688	7.274

5.0 SSREFLECT MODE IN HOL LIGHT

This chapter describes our implementation of a special proof language SSReflect [GMT11] in HOL Light. We start with a short overview of formal proof writing techniques. Then we briefly discuss the SSReflect language and its implementation in HOL Light. We end the chapter with a demonstration of some results formalized in SSReflect mode in HOL Light. An introduction to SSReflect language can be found in [GM11]. A reference manual for SSReflect mode in HOL Light is available [Sol12a].

5.1 FORMAL PROOF TECHNIQUES

Two major techniques for writing formal proofs are declarative and procedural approaches. The former approach is the only way to produce formal proofs in one of the oldest proof assistants Mizar [Miz]. Procedural proofs are ubiquitous in the family of HOL proof assistants (HOL Light [Har10], HOL4 [Hol], Isabelle/HOL [Isa], PVS [PVS]) and in proof assistants based on type theory (Coq [BC04, Coq], Twelf [Twe]). Moreover, many proof assistants support different proof modes. For instance, the Isar language [Nip03] allows writing declarative proofs in Isabelle. The `miz3` interface developed by F. Wiedijk [Wie01, Wie12b] provides declarative proofs for HOL Light users. It is based on an earlier work by J. Harrison [Har96b]. A more comprehensive discussion of declarative and procedural proof styles can be found in [Har96c].

Declarative proofs resemble usual mathematical proofs. A declarative proof consists of steps where each step includes an explicit goal which must be proved at that step. After this explicit goal, a list of references of proved results is given. The goal must be derived automatically from these references and basic logical rules. Declarative proofs may also include some auxiliary steps. Usually, steps in a declarative proof are not very big since an automation procedure cannot resolve difficult goals. Declarative proofs may be written from the beginning to the end without any interaction with a proof assistant. When a proof is ready, then the proof assistant checks it and if it finds any errors, then the proof must be corrected. Most errors (we don't count typos here) are inference errors when the proof assistant cannot derive next step from given references. In that case, the proof should be refined. An advantage of such an approach is that it is possible to write formal proofs in an iterative way: first of all, a proof skeleton is created [Wie02] and then this skeleton is refined until a proof assistant can accept the proof.

An example of a declarative proof in the `miz3` language [Wie01] of the fact that each infinite set contains an element is given below.

```
!s. INFINITE s ==> ?x:A. x IN s
proof
  let s be A->bool;
  assume INFINITE s;
  ~(s = {}) by INFINITE_NONEMPTY;
  consider x such that
    ~(x IN s <=> x IN {}) [1] by EXTENSION;
  take x;
  ~(x IN {}) by NOT_IN_EMPTY;
qed by 1
```

(This proof is taken from the file `miz3/Samples/samples.ml` in the HOL Light distribution [Har10].)

Procedural proofs work differently. A user states a goal and then applies tactics (special transformation procedures) to the current goal. The main idea is to apply tactics which potentially simplify the current goal and produce simpler subgoals. Sometimes, it is required to state a new subgoal explicitly (like in declarative proofs) which is allowed in procedural proofs. Procedural proofs are usually shorter than the corresponding declarative proofs.

They are developed in an interactive mode when each tactic is run by a user and the result is immediately returned by a proof assistant. Procedural proofs are not human-readable but it is not difficult to run them step by step to understand their logic or to use special tools which produce corresponding declarative proofs or informal human-readable proofs [HMBC99].

The following example shows a HOL Light procedural proof [Har10] corresponding to the declarative proof given above.

```
prove(‘!s. INFINITE s ==> ?x:A. x IN s‘,
  REPEAT STRIP_TAC THEN
    REWRITE_TAC[MEMBER_NOT_EMPTY] THEN
    MATCH_MP_TAC INFINITE_NONEMPTY THEN
    ASM_REWRITE_TAC[]);;
```

This proof is shorter than the declarative proof above. On the other hand, it is difficult to read this procedural proof and it is necessary to run it step by step in order to understand its logic.

Another important difference between proof assistants and proof languages is how much automation is used when one writes formal proofs. All declarative system rely on a sufficiently well-developed automation. Though, this automation may be deliberately weakened in order to keep control over semantics of each proof step. Isabelle system is an example of a proof assistant with very powerful automation procedures [BN10]. On the other hand, SSReflect language for Coq is an example of a system where automation is limited but the proofs are still concise. HOL Light has several powerful automation procedures [Har96a, MH05] but most HOL Light proofs do not rely on the automation too much. It is not easy to say how much one should rely on automation. On one hand, it is tedious to prove trivial results which can be easily eliminated by automatic procedures. On the other hand, there are no universal automatic procedure which can help in all (even trivial) cases. Linear arithmetic procedures can easily deal with relatively large goals of linear equalities and inequalities but even a trivial nonlinear operation could completely stop them. The Robbins conjecture is the only major result proved by a fully automated procedure [McC97]. However, this result was possible only because of the nature of the conjecture itself.

The last issues with proof assistants is portability of results. It is desirable to be able to use results from one proof assistant in another. In order to achieve this, one needs to be able to translate both the statement of a theorem and its proof. The main issue is that different proof assistants are based on different principles (e.g., type theory in Coq and higher order logic in HOL Light). There are two basic approaches: low level and high level translations. Low level translators produce new proofs by translating all elementary steps in the original proof. Usually, it is not difficult to find corresponding elementary steps in different proof assistants. The problem is that proofs with basic inference steps quickly become very large. Moreover, even a small formal lemma could rely on a huge library of other results. So it is often necessary to translate large libraries of results which could be not compatible with existing libraries. Nevertheless, several successful low-level translators exist [Wie, KW10, OS06].

Carefully designed high level proof languages could produce proofs for different proof assistants. These languages must not depend on special properties of proof assistants for which they work. Declarative proofs are already proofs in high level formal languages which can be translated from one system into another. (It is important to use pure declarative constructions even if the corresponding language allows direct procedural statements.)

A good analogy is the situation with low level and high level programming languages. In very rare cases, one translates low level programs from one system into another. On the other hand, most programs written in high level programming languages are compatible with different systems.

It is mostly a matter of taste which formal proof approach to use. Some people prefer declarative proofs, others like the procedural approach. The same situation is with automation: for trivial domains with a good automation (linear arithmetic) there is no point to ignore it, for more general results it is up to a user whether to use automation or not. Portability is very important when one starts a large formalization project: either everything should be done in one proof assistant or it is required to consider how results can be checked in different proof assistants. There is an issue with Flyspeck project where an important formalization step—generation of all contravening hypermaps—is done in Isabelle [NBS06] meanwhile all other results (including the theory of hypermaps) are in HOL Light.

5.2 SSREFLECT/HOL LIGHT

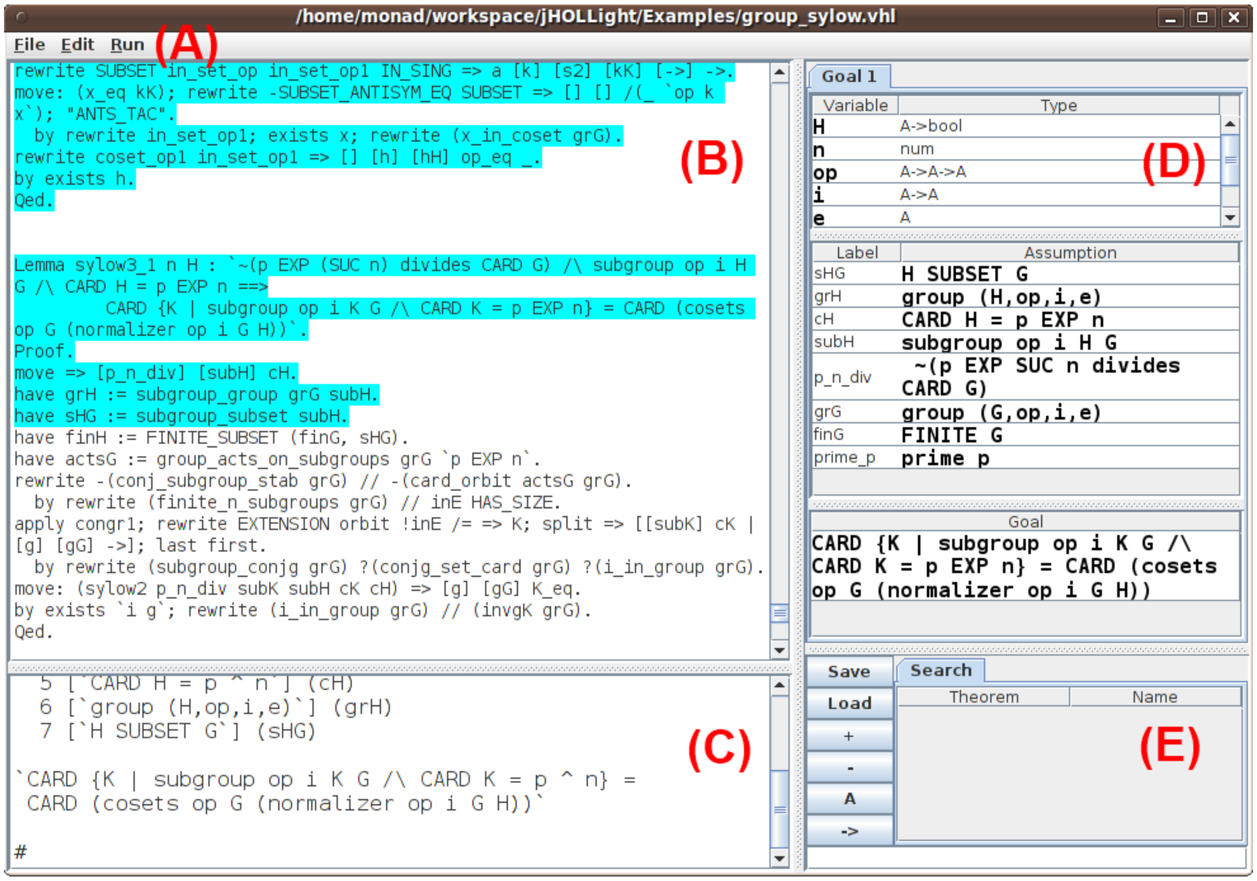
SSReflect language was developed by G. Gonthier at Microsoft Research–INRIA to produce mathematical proofs in the Coq proof assistant [GMT11, GGMR09, GMR07]. This language was very successfully applied to a recent formal proof of the Odd Order Theorem [Gon12, BG94, Pet00]. The author of this work also made a small contribution to that formalization project by formalizing chapter 7 of [Pet00].

SSReflect/HOL Light is our implementation of SSReflect language for the HOL Light proof assistant. The main idea behind development of SSReflect mode in HOL Light was to provide new opportunities for creating formal proofs in HOL Light. Another goal was to port some results from SSReflect/Coq library into HOL Light. Our implementation of SSReflect/HOL Light is not completely compatible with original SSReflect/Coq. Moreover, the syntax of formal terms is different since one system is based on Coq and another on HOL Light. Nevertheless, several important results and theorems have been successfully formalized in SSReflect/HOL Light and some SSReflect/Coq libraries have been ported into HOL Light (see Section 5.3).

SSReflect/HOL Light is implemented in OCaml [Cam] and Java [Jav] programming languages. The OCaml part of the implementation contains new HOL Light tactics and extensions for working with locally defined variables and assumptions. The Java part consists of a user interface and a translator of SSReflect commands and tactics into OCaml and HOL Light commands.

The SSReflect/HOL Light user interface is shown at Figure 1. The main parts of the interface are: the main menu (A), the text editing area (B), the message area (C), the proof status area (D) which includes the list of context variable and the tabbed list of subgoals with assumptions, and the search area (E). A detailed description of the user interface and SSReflect/HOL Light commands and tactics can be found in the user manual [Sol12a]. Below, we present highlights of SSReflect mode in HOL Light.

- The SSReflect/HOL Light user interface is convenient for an interactive proof development. A user can easily move to any point in a proof script, make one proof step or return back. A special window always shows the current goal state, all assumptions and free



```
rewrite SUBSET in_set_op in_set_op1 IN_SING => a [k] [s2] [kK] [->] ->.
move: (x_eq kK); rewrite -SUBSET_ANTISYM_EQ SUBSET => [] [] /(_ `op k
x`); "ANTS_TAC".
  by rewrite in_set_op1; exists x; rewrite (x_in_coset grG).
rewrite coset_op1 in_set_op1 => [] [h] [hH] op_eq _.
by exists h.
Qed.


Lemma sylow3_1 n H : `~(p EXP (SUC n) divides CARD G) /\ subgroup op i H
G /\ CARD H = p EXP n ==>
         CARD {K | subgroup op i K G /\ CARD K = p EXP n} = CARD (cosets
op G (normalizer op i G H))`.
Proof.
move => [p_n_div] [subH] cH.
have grH := subgroup_group grG subH.
have sHG := subgroup_subset subH.
have finH := FINITE_SUBSET (finG, sHG).
have actsG := group_acts_on_subgroups grG `p EXP n`.
rewrite -(conj_subgroup_stab grG) // -(card_orbit actsG grG).
  by rewrite (finite_n_subgroups grG) // inE HAS_SIZE.
apply congr1; rewrite EXTENSION orbit !inE /= => K; split => [[subK] cK |
[g] [gG] ->]; last first.
  by rewrite (subgroup_conjg grG) ?(conjg_set_card grG) ?(i_in_group grG).
move: (sylow2 p_n_div subK subH cK cH) => [g] [gG] K_eq.
by exists `i g`; rewrite (i_in_group grG) // (invgK grG).
Qed.
```

```
  5 [`CARD H = p ^ n`] (cH)
  6 [`group (H,op,i,e)`] (grH)
  7 [`H SUBSET G`] (sHG)

`CARD {K | subgroup op i K G /\ CARD K = p ^ n} =
 CARD (cosets op G (normalizer op i G H))`

#
```

Variable	Type
H	A->bool
n	num
op	A->A->A
i	A->A
e	A

Label	Assumption
sHG	H SUBSET G
grH	group (H,op,i,e)
cH	CARD H = p EXP n
subH	subgroup op i H G
p_n_div	~(p EXP SUC n divides CARD G)
grG	group (G,op,i,e)
finG	FINITE G
prime_p	prime p

Goal
CARD {K | subgroup op i K G /\ CARD K = p EXP n} = CARD (cosets op G (normalizer op i G H))

Theorem	Name

Figure 1: SSReflect/HOL Light user interface

variables. Searching existing theorems is simple and all found results can be memorized in a special table.

- Any SSReflect/HOL Light script file can be translated into a HOL Light file which can be loaded directly in HOL Light. It is only required to load two small SSReflect supporting files written in OCaml.
- A special section mechanism helps to organize lemmas and theorems which share the same variables and assumptions. If a local variable is declared inside a section, then it can be used as a free variable in statements of lemmas inside the section. All local variables will be generalized in statements of lemmas when the section is closed. Local assumptions are also available. All lemmas in a section proved after a given local assumption share this assumption. When the section is closed, then all hypotheses will be discharged for all lemmas proved in that section.
- All HOL Light commands and tactics are available in SSReflect/HOL Light.
- A very convenient way to introduce and discharge variables and assumptions.
- SSReflect/HOL Light provides a convenient way for constructing theorems by "applying" a theorem or a term to another theorem. The syntax is `(th arg1 ... argn)`. The "application" works as follows. The expression `(th arg1 arg2)` is the same as `((th arg1) arg2)`. If a theorem has the form `th = |- P ==> Q` then `arg1` must be a theorem of the form `|- P'` such that it is possible to match `P'` and `P`. The result of `(th arg1)` is `|- Q`. If a theorem has the form `th = |- !x. P x`, then the argument must be a term `` `y` ``, and `(th arg1)` will be `|- P y` (it is assumed that the type of `` `y` `` is correct). In fact, the construction `(th arg)` is more general than the standard HOL Light rules `MATCH_MP` and `ISPEC`. The command `(th arg)` matches `arg` against all assumptions of `th` until it finds a good match. For instance, suppose `th = |- A /\ B ==> C`, `th_a = |- A`, and `th_b = |- B`, then the expression `(th th_a)` yields `|- B ==> C`, `(th th_b) = |- A ==> C`, and `(th th_b th_a) = |- C`.
- SSReflect/HOL Light has a powerful rewriting mechanism. SSReflect `rewrite` tactic works in a similar way as the standard `REWRITE_TAC[]` in HOL Light for simple equational theorems. But it also accepts an equational theorem with conditions. If `th = |- P ==> (f = g)`, then `rewrite th` changes all occurrences of `f` to `g` in a goal

and introduces a new subgoal `P`. There are other options of `rewrite` which help to select a subterm for rewriting or control the number of rewrites with one or several theorems.

5.3 RESULTS

All main theoretical results for verification of nonlinear inequalities were proved with SSReflect/HOL Light. Two theorems of the Flyspeck project were formalized in SSReflect/HOL Light (theorems with labels `FNJLBXS` and `FCDJDOT`, see [Hal12a]).

Two basic SSReflect/Coq libraries were ported to HOL Light with SSReflect/HOL Light. These libraries are `ssrnat.v` and `seq.v`. They contain a lot of useful definitions and theorems for natural numbers and sequences. The translation process of these libraries was not automatic but it was simple and proofs of some results (even most difficult ones) were translated almost automatically. For instance, consider the following proof of a lemma about rotations of a sequence. The original SSReflect/Coq proof is

```
Lemma rot_rot m n s : rot m (rot n s) = rot n (rot m s).
Proof.
case: (ltnP (size s) m) => Hm.
  by rewrite !(@rot_oversize T m) ?size_rot 1?ltnW.
case: (ltnP (size s) n) => Hn.
  by rewrite !(@rot_oversize T n) ?size_rot 1?ltnW.
by rewrite !rot_add_mod 1?addnC.
Qed.
```

The translated SSReflect/HOL Light proof is

```
Lemma rot_rot m n s : `rot m (rot n s) = rot n (rot m s)`.
Proof.
case: (ltnP `sizel s` m) => Hm.
  by rewrite ![`rot m _`]rot_oversize ?size_rot 1?ltnW.
case: (ltnP `sizel s` n) => Hn.
```

```
  by rewrite ![`rot n _`]rot_oversize ?size_rot 1?ltnW.
by rewrite !rot_add_mod 1?addnC.
Qed.
```

As we can see, both proofs are almost identical.

HOL Light is not very friendly for formalizing abstract algebra. However, a formalization of Sylow theorem has been successfully done with SSReflect/HOL Light. As far as we know, this is the most advanced abstract algebraic result formalized in HOL Light (see a list of some important theorems formalized in different proof assistants [Wie12a]). The formalization of Sylow theorems and all necessary group theoretic results is relatively short (about 2000 lines in SSReflect/HOL Light); and Sylow theorems themselves take less than 150 lines. The proof of Sylow theorems is based on the proof given in the notes [Con12].

Here is the statement of the first Sylow theorem in SSReflect/HOL Light:

```
Variable G : `:A->bool`.
Variable op : `:A->A->A`.
Variable i : `:A->A`.
Variable e : `:A`.
Variable p : `:num`.

Hypothesis grG : `group (G,op,i,e)`.
Hypothesis finG : `FINITE G`.
Hypothesis prime_p : `prime p`.

Lemma sylow1 :
   `!n. p EXP n divides CARD G
       ==> ?H. subgroup op i H G /\ CARD H = p EXP n`.
Proof. ... Qed.
```

The first complete formalization of Sylow theorems is not very old [TR06] and it was done with an early version of SSReflect in Coq.

APPENDIX

SOURCE CODE ORGANIZATION

The appendix contains descriptions of all source code files and their location in the Flyspeck repository [Hal12b]. All information is valid for revision 2991. Each section of the appendix describes the source code of the corresponding chapter.

A.1 FORMAL ARITHMETIC

Base directory: `formal_ineqs/arith`

File(s)	Description	Back Refs
`../arith_options.hl`	Options of the formal arithmetic library.	
`arith_num.hl`	Formal natural number arithmetic with an arbitrary base.	2.1.2
`arith_cache.hl`	Cached natural number arithmetic.	2.2.4
`nat.hl`	Main file of the natural number arithmetic library.	2.1
`num_exp_theory.hl`	Theory of the exponential representation of natural numbers.	2.2.2
`float_theory.hl`	Basic definitions and theory of floating-point numbers.	2.2.3

`interval_arith.hl`	Basic theory of interval arithmetic.	2.4.1
`float.hl`	Operations with floating-point numbers and interval arithmetic.	2.2.3, 2.4.2
`more_float.hl`	Additional floating-point operations.	
`float_atn.hl`	Floating-point interval arctangent and arccosine.	2.4.3
`eval_interval.hl`	Interval evaluation of general HOL Light terms.	2.4.4

A.2 FORMAL VERIFICATION OF LINEAR PROGRAMS

Base directory: `formal_lp/more_arith`

File(s)	**Description**	**Back Refs**
`arith_int.hl`	Formal integer arithmetic.	2.3
`lin_f.hl`	Theory of linear functions.	3.2
`prove_lp.hl`	Formal verification procedure of bounds of general linear programs.	3.2

Base directory: `formal_lp/hypermap`

File(s)	**Description**	**Back Refs**
`constants_approx.hl`	Interval approximations of some constants.	
`list_conversions.hl`	General list conversions	
`list_hypermap_defs.hl`	Definitions for hypermaps represented as lists of numbers.	3.3
`list_hypermap.hl`	Theory of hypermaps represented as lists of numbers.	3.3

`list_hypermap_iso.hl`	Isomorphism between a fan hypermap and a hypermap represented as a list of numbers.	3.3
`list_hypermap_computations.hl`	Formal computations of elements of hypermaps represented as lists of numbers.	3.3
`contravening_ineqs.hl`	Generation of linear constraints from general theorems.	3.3
`nobranching_lp.hl`	Main verification procedure for non-branching Flyspeck linear programs.	3.2, 3.3

The directory `formal_lp/LP_HL` contains C# code for translating `GLPK` solutions into input files for formal verification procedures.

A.3 NONLINEAR INEQUALITIES

Base directory: `formal_ineqs`

File(s)	**Description**	**Back Refs**
`arith_options.hl`	Options of the formal arithmetic library.	
`verifier_options.hl`	General options of verification procedures.	
`examples*.hl`	Examples and tests.	4.7
`arith/*.hl`	Formal arithmetic libraries.	2
`informal/*.hl`	Informal verification procedures.	4.5
`lib/*.hl`	Supporting libraries.	
`list/*.hl`	General and special list conversions.	
`misc/*.hl`	Miscellaneous functions and definitions.	
`taylor/theory/*.vhl`	SSReflect/HOL Light theory files of Taylor approximations.	4.2

taylor/*.hl	Formal computations of Taylor intervals.	4.3
verifier/*.hl	Formal verification procedures.	4.4
verifier/interval_m/*.ml	Solution certificate search procedures.	4.1.2

A.4 SSREFLECT MODE IN HOL LIGHT

Base directory: jHOLLight

File(s)	**Description**	**Back Refs**
caml/raw_printer.hl	OCaml/Java communication functions.	5.2
caml/sections.hl	Local definitions and assumptions in HOL Light.	5.2
caml/ssreflect.hl	New HOL Light tactics.	5.2
Examples/*.vhl	SSReflect/HOL Light sample files (including Sylow theorems).	5.3
src/**/*.java	Java source code files.	5.2

Base directory: jHOLLight/src/edu/pitt/math/jhol

File(s)	**Description**	**Back Refs**
caml/*.java	OCaml objects in Java.	5.2
core/*.java	HOL Light objects in Java.	5.2
core/parser/*.java	Parsing of OCaml/HOL Light output.	5.2
core/printer/*.java	Printing of HOL Light terms and theorems in Java.	5.2
gui/*.java	GUI components.	5.2
ssreflect/gui/*.java	SSReflect/HOL Light GUI.	5.2
ssreflect/parser/**/*.java	SSReflect/HOL Light parsing and translating functions.	5.2

BIBLIOGRAPHY

[AGST10] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with imperative features and its application to SAT verification. In *Proceedings of the First international conference on Interactive Theorem Proving*, ITP'10, pages 83–98, Berlin, Heidelberg, 2010. Springer-Verlag.

[AMP] AMPL modeling language for mathematical programming. http://www.ampl.com.

[AP08] Behzad Akbarpour and Lawrence Paulson. MetiTarski: An automatic prover for the elementary functions. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki, and Freek Wiedijk, editors, *Intelligent Computer Mathematics*, volume 5144 of *Lecture Notes in Computer Science*, pages 217–231. Springer Berlin / Heidelberg, 2008.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[BG94] H. Bender and G. Glauberman. *Local analysis for the odd order theorem*, volume 188 of *LMS*. Cambridge University Press, 1994.

[BN10] Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement Day. In J. Giesl and R. Hähnle, editors, *Automated Reasoning (IJCAR 2010)*, volume 6173 of *LNCS*, pages 107–121. Springer, 2010.

[Cam] The Caml Language. http://caml.inria.fr/.

[Col75] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Automata theory and formal languages (Second GI Conf., Kaiserslautern, 1975)*, pages 134–183. Lecture Notes in Comput. Sci., Vol. 33. Springer, Berlin, 1975.

[Con12] Keith Conrad. The Sylow Theorems, 2012. http://www.math.uconn.edu/~kconrad/blurbs/grouptheory/sylowpf.pdf.

[Coq] The Coq proof assistant. http://coq.inria.fr/.

[DEC] decimal (C# Reference). http://msdn.microsoft.com/en-us/library/364x0z75.aspx.

[DLM09] Marc Daumas, David Lester, and César Muñoz. Verified real number calculations: A library for interval arithmetic. *IEEE Transactions on Computers*, 58(2):226–237, February 2009.

[Fel] Joel Feldman. Equality of mixed partial derivatives. http://www.math.ubc.ca/~feldman/m200/mixed.pdf.

[GGMR09] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging Mathematical Structures. In Tobias Nipkow and Christian Urban, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, Munich, Allemagne, 2009. Springer.

[GLP] GLPK (GNU Linear Programming Kit). http://www.gnu.org/software/glpk/.

[GM11] G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. http://hal.inria.fr/inria-00515548/, 2011.

[GMR07] G. Gonthier, A. Mahboubi, and L. Rideau. *A modular formalisation of finite group theory*, volume 4732 of *LNCS*, pages 86–101. Springer, 2007.

[GMT11] G. Gonthier, A. Mahboubi, and E. Tassi. A small scale reflection extension for the Coq system. http://hal.archives-ouvertes.fr/inria-00258384/, 2011.

[GNSW07] Herman Geuvers, Milad Niqui, Bas Spitters, and Freek Wiedijk. Constructive analysis, types and exact real numbers (overview article). *Mathematical Structures in Computer Science*, 17(1):3–36, 2007.

[Gol90] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1990.

[Gon12] G. Gonthier. Mathematical Components, 2012. http://www.msr-inria.inria.fr/Projects/math-components.

[GT06] Benjamin Grégoire and Laurent Théry. A purely functional library for modular arithmetic and its application to certifying large prime numbers. In *Proceedings of the Third international joint conference on Automated Reasoning*, IJCAR'06, pages 423–437, Berlin, Heidelberg, 2006. Springer-Verlag.

[Hal03] T. C. Hales. Some algorithms arising in the proof of the Kepler conjecture. *Discrete and computational geometry*, 25:489–507, 2003.

[Hal06a] T. C. Hales. Introduction to the Flyspeck project. In Thierry Coquand, Henri Lombardi, and Marie-Françoise Roy, editors, *Mathematics, Algorithms, Proofs*, number 05021 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2006/432.

[Hal06b] T. C. Hales. An overview of the Kepler conjecture. *Discrete and Computational Geometry*, 36(1):5–20, 2006.

[Hal07] T. C. Hales. The Jordan curve theorem, formally and informally. *American Math. Monthly*, 114(10):882–894, December 2007.

[Hal08] T. C. Hales. Formal proof. *Notices of the AMS*, 55(11):1370–1380, December 2008.

[Hal10] T. C. Hales. Linear programs for the Kepler Conjecture. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software – ICMS 2010*. Springer, September 2010.

[Hal12a] T. C. Hales. *Dense Sphere Packings: a blueprint for formal proofs*, volume 400 of *London Math Soc. Lecture Note Series*. Cambridge University Press, 2012.

[Hal12b] T. C. Hales. The Flyspeck Project, 2012. http://code.google.com/p/flyspeck.

[Har] John Harrison. *The HOL Light manual*. http://www.cl.cam.ac.uk/~jrh13/hol-light/reference.pdf.

[Har95] J. Harrison. Inductive definitions: automation and application. In Phillip J. Windley, Thomas Schubert, and Jim Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, pages 200–213, Aspen Grove, Utah, 1995. Springer-Verlag.

[Har96a] J. Harrison. Optimizing proof search in model elimination. In *Proceedings of the 13th International Conference on Automated Deduction (CADE-13)*, volume 1104 of *LNCS*, pages 313–327. Springer, 1996.

[Har96b] John Harrison. A Mizar mode for HOL. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 203–220, Turku, Finland, 1996. Springer-Verlag.

[Har96c] John Harrison. Proof style. In Eduardo Giménex and Christine Pausin-Mohring, editors, *Types for Proofs and Programs: International Workshop TYPES'96*, volume 1512 of *Lecture Notes in Computer Science*, pages 154–172, Aussois, France, 1996. Springer-Verlag.

[Har98] John Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[Har99] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, Nice, France, 1999. Springer-Verlag.

[Har05] John Harrison. A HOL theory of Euclidean space. In *Theorem proving in higher order logics*, volume 3603 of *Lecture Notes in Comput. Sci.*, pages 114–129. Springer, Berlin, 2005.

[Har06a] John Harrison. Floating-point verification using theorem proving. In Marco Bernardo and Alessandro Cimatti, editors, *Formal Methods for Hardware Verification, 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2006*, volume 3965 of *Lecture Notes in Computer Science*, pages 211–242, Bertinoro, Italy, 2006. Springer-Verlag.

[Har06b] John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Proceedings of the third International Joint Conference, IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191, Seattle, WA, 2006. Springer-Verlag.

[Har07] J. Harrison. Verifying nonlinear real formulas via sums of squares. In K. Schneider and J. Brandt, editors, *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 102–118. Springer-Verlag, Kaiserslautern, Germany, 2007.

[Har08] J. Harrison. Formal proof – theory and practice. *Notices of the AMS*, 55(11):1395–1406, December 2008.

[Har10] J. Harrison. The HOL Light theorem prover, 2010. http://www.cl.cam.ac.uk/~jrh13/hol-light/index.html.

[Has] The Haskell programming language. http://www.haskell.org.

[HF06] T. C. Hales and S. P. Ferguson. The Kepler conjecture. *Discrete and Computational Geometry*, 36(1):1–269, 2006.

[HHM$^+$09] T. C. Hales, J. Harrison, S. McLaughlin, T. Nipkow, S. Obua, and R. Zumkeller. A revision of the proof of the Kepler Conjecture. *DCG*, 2009.

[HMBC99] Amanda M. Holland-Minkley, Regina Barzilay, and Robert L. Constable. Verbalization of high-level formal proofs. In *Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence*, AAAI '99/IAAI '99, pages 277–284, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[Hol] HOL4. http://hol.sourceforge.net.

[IEE85] IEEE Standards Committee 754. *IEEE Standard for binary floating-point arithmetic, ANSI/IEEE Standard 754-1985.* Institute of Electrical and Electronics Engineers, New York, 1985.

[Isa] Isabelle. http://isabelle.in.tum.de/.

[Jav] Java programming language. http://www.java.com/.

[Jul08] Nicolas Julien. Certified exact real arithmetic using co-induction in arbitrary integer base. In Jacques Garrigue and ManuelV. Hermenegildo, editors, *Functional and Logic Programming*, volume 4989 of *Lecture Notes in Computer Science*, pages 48–63. Springer Berlin Heidelberg, 2008.

[Kar95] A.A. Karatsuba. The complexity of computations. *Proceedings of the Steklov Institute of Mathematics-Interperiodica Translation*, 211:169–183, 1995.

[Kea96] R. B. Kearfott. Interval computations: Introduction, uses, and resources. *Euromath Bulletin*, 2:95–112, 1996.

[KW10] Chantal Keller and Benjamin Werner. Importing HOL Light into Coq. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 2010.

[Loa98] Ralph Loader. Notes on simply typed lambda calculus, 1998. http://www.lfcs.inf.ed.ac.uk/reports/98/ECS-LFCS-98-381/.

[LPS] Mixed Integer Linear Programming (MILP) solver. http://sourceforge.net/projects/lpsolve/.

[McC97] W. McCune. Solution of the Robbins problem. *JAR*, 19(3):263–276, 1997.

[MH05] S. McLaughlin and John Harrison. A proof-producing decision procedure for real arithmetic. In *Automated deduction—CADE-20*, volume 3632 of *Lecture Notes in Comput. Sci.*, pages 295–314. Springer, Berlin, 2005.

[Miz] Mizar home page. http://mizar.org/.

[MN12] César Muñoz and Anthony Narkawicz. Formalization of a representation of Bernstein polynomials and applications to global optimization. *Journal of Automated Reasoning*, 2012. Accepted for publication.

[NBS06] T. Nipkow, G. Bauer, and P. Schultz. Flyspeck I: Tame Graphs. In Ulrich Furbach and Natarajan Shankar, editors, *International Joint Conference on Au-*

tomated Reasoning, volume 4130 of *Lect. Notes in Comp. Sci.*, pages 21–35. Springer-Verlag, 2006.

[Nip03] Tobias Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *Lect. Notes in Comp. Sci.*, pages 259–278. Springer-Verlag, 2003.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[Obu05] S. Obua. Proving bounds for real linear programs in Isabelle/HOL. In J. Hurd and T. F. Melham, editors, *Theorem Proving in Higher Order Logics*, volume 3603 of *Lect. Notes in Comp. Sci.*, pages 227–244. Springer-Verlag, 2005.

[Obu08] Steven Obua. *Flyspeck II: The Basic Linear Programs.* PhD thesis, Technische Universität München, 2008.

[O'C08] Russell O'Connor. Certified exact transcendental real number computation in Coq. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 246–261, Berlin, Heidelberg, 2008. Springer-Verlag.

[O'C09] Russell O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* PhD thesis, Radboud University Nijmegen, 2009.

[OS06] Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In *Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 2006.

[OT09] Steven Obua and Tobias. Flyspeck II: the basic linear programs. *Annals of Mathematics and Artificial Intelligence*, 56(3–4), 2009.

[Pau12] Lawrence Paulson. MetiTarski: Past and future. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, volume 7406 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin / Heidelberg, 2012.

[Pet00] T. Peterfalvi. *Character theory for the odd order theorem*, volume 272 of *LMS*. Cambridge University Press, 2000.

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

[SH11] A. Solovyev and T. C. Hales. *Efficient formal verification of bounds of linear programs*, volume 6824 of *LNCS*, pages 123–132. Springer-Verlag, 2011.

[Sol12a] Alexey Solovyev. SSReflect/HOL Light reference manual (v1.1), 2012. http://flyspeck.googlecode.com/files/HOL-SSReflect%20v1.1.pdf.

[Sol12b] Alexey Solovyev. A tool for formal verification of nonlinear inequalities, 2012. http://flyspeck.googlecode.com/files/FormalVerifier.pdf.

[Tar51] A. Tarski. *A decision method for elementary algebra and geometry*. University of California Press, Berkeley and Los Angeles, Calif., 1951. 2nd ed.

[TR06] Laurent Thery and Laurence Rideau. Formalising Sylow's theorems in Coq. Technical Report RT-0327, INRIA, 2006.

[Twe] The Twelf Project. http://twelf.org.

[Wie] Freek Wiedijk. Encoding the HOL Light logic in Coq. http://www.cs.ru.nl/~freek/notes/holl2coq.pdf.

[Wie01] Freek Wiedijk. Mizar Light for HOL Light. In *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '01, pages 378–394, London, UK, UK, 2001. Springer-Verlag.

[Wie02] Freek Wiedijk. Formal proof sketches. In *In Proceedings of TYPES03, volume 3085 of LNCS*, pages 378–393. Springer, 2002.

[Wie06] Freek Wiedijk, editor. *The seventeen provers of the world*, volume 3600 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2006. Foreword by Dana S. Scott, Lecture Notes in Artificial Intelligence.

[Wie08] F. Wiedijk. Formal proof – getting started. *Notices of the AMS*, 55(11):1408–1414, December 2008.

[Wie12a] F. Wiedijk. Formalizing 100 theorems. http://www.cs.ru.nl/~freek/100/, referenced 2012.

[Wie12b] Freek Wiedijk. A synthesis of the procedural and declarative styles of interactive theorem proving. *Logical Methods in Computer Science*, 8(1), 2012.

[Zum06] Roland Zumkeller. Formal global optimisation with Taylor models. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 408–422. Springer Berlin / Heidelberg, 2006.

[Zum08] Roland Zumkeller. *Global Optimization in Type Theory*. PhD thesis, École Polytechnique Paris, 2008.

[Zum09] R. Zumkeller. Sergei. A Global Optimization Tool, 2009. http://code.google.com/p/sergei/.