axiom™

The 30 Year Horizon

Manuel Bronstein *William Burge* *Timothy Daly*
James Davenport *Michael Dewar* *Martin Dunstan*
Albrecht Fortenbacher *Patrizia Gianni* *Johannes Grabmeier*
Jocelyn Guidry *Richard Jenks* *Larry Lambe*
Michael Monagan *Scott Morrison* *William Sit*
Jonathan Steinbach *Robert Sutor* *Barry Trager*
Stephen Watt *Jim Wen* *Clifton Williamson*

Volume 13: Proving Axiom Correct

Inclusion of names in the list of credits is based on historical information and is as accurate as possible. Inclusion of names does not in any way imply an endorsement but represents historical influence on Axiom development.

Michael Albaugh	Cyril Alberga	Roy Adler
Christian Aistleitner	Richard Anderson	George Andrews
S.J. Atkins	Henry Baker	Martin Baker
Stephen Balzac	Yurij Baransky	David R. Barton
Thomas Baruchel	Gerald Baumgartner	Gilbert Baumslag
Michael Becker	Nelson H. F. Beebe	Jay Belanger
David Bindel	Fred Blair	Vladimir Bondarenko
Mark Botch	Raoul Bourquin	Alexandre Bouyer
Karen Braman	Peter A. Broadbery	Martin Brock
Manuel Bronstein	Stephen Buchwald	Florian Bundschuh
Luanne Burns	William Burge	Ralph Byers
Quentin Carpent	Robert Caviness	Bruce Char
Ondrej Certik	Tzu-Yi Chen	Cheekai Chin
David V. Chudnovsky	Gregory V. Chudnovsky	Mark Clements
James Cloos	Jia Zhao Cong	Josh Cohen
Christophe Conil	Don Coppersmith	George Corliss
Robert Corless	Gary Cornell	Meino Cramer
Jeremy Du Croz	David Cyganski	Nathaniel Daly
Timothy Daly Sr.	Timothy Daly Jr.	James H. Davenport
David Day	James Demmel	Didier Deshommes
Michael Dewar	Jack Dongarra	Jean Della Dora
Gabriel Dos Reis	Claire DiCrescendo	Sam Dooley
Lionel Ducos	Iain Duff	Lee Duhem
Martin Dunstan	Brian Dupee	Dominique Duval
Robert Edwards	Heow Eide-Goodman	Lars Erickson
Richard Fateman	Bertfried Fauser	Stuart Feldman
John Fletcher	Brian Ford	Albrecht Fortenbacher
George Frances	Constantine Frangos	Timothy Freeman
Korrinn Fu	Marc Gaetano	Rudiger Gebauer
Van de Geijn	Kathy Gerber	Patricia Gianni
Gustavo Goertkin	Samantha Goldrich	Holger Gollan
Teresa Gomez-Diaz	Laureano Gonzalez-Vega	Stephen Gortler
Johannes Grabmeier	Matt Grayson	Klaus Ebbe Grue
James Griesmer	Vladimir Grinberg	Oswald Gschnitzer
Ming Gu	Jocelyn Guidry	Gaetan Hache
Steve Hague	Satoshi Hamaguchi	Sven Hammarling
Mike Hansen	Richard Hanson	Richard Harke
Bill Hart	Vilya Harvey	Martin Hassner
Arthur S. Hathaway	Dan Hatton	Waldek Hebisch
Karl Hegbloom	Ralf Hemmecke	Henderson
Antoine Hersen	Roger House	Gernot Hueber
Pietro Iglio	Alejandro Jakubi	Richard Jenks
William Kahan	Kyriakos Kalorkoti	Kai Kaminski

Grant Keady
Ted Kosan
Bernhard Kutzler
Kaj Laurson
Frederic Lehobey
Ren-Cang Li
Richard Luczak
Alasdair McAndrew
Edi Meier
Victor S. Miller
H. Michael Moeller
Scott Morrison
William Naylor
John Nelder
Jinzhong Niu
Kostas Oikonomou
Bill Page
Michel Petitot
Frederick H. Pitts
Claude Quitte
Anatoly Raportirenko
Guilherme Reis
Jean Rivlin
Raymond Rogers
Philip Santas
Gerhard Schneider
Frithjof Schulze
V. Sima
Elena Smirnova
Christine Sundaresan
Eugene Surowitz
James Thatcher
Dylan Thurston
Themos T. Tsikas
Stephen Watt
M. Weller
Thorsten Werther
James T. Wheeler
Clifton J. Williamson
Robert Wisbauer
Knut Wolf
Clifford Yapp
Richard Zippel
Dan Zwillinger

Wilfrid Kendall
Paul Kosinski
Tim Lahey
George L. Legendre
Michel Levaud
Rudiger Loos
Camm Maguire
Bob McElrath
Ian Meikle
Gerard Milmeister
Michael Monagan
Joel Moses
Patrice Naudin
Godfrey Nolan
Michael O'Connor
Humberto Ortiz-Zuazaga
David Parnas
Didier Pinchon
Jose Alfredo Portes
Arthur C. Ralfs
Albert D. Rich
Huan Ren
Nicolas Robidoux
Michael Rothstein
Alfred Scheerhorn
Martin Schoenert
Fritz Schwarz
Nick Simicich
Jonathan Steinbach
Robert Sutor
Max Tegmark
Balbir Thomas
Steve Toleque
Gregory Vanuxem
Jaap Weel
Mark Wegman
Michael Wester
John M. Wiley
Stephen Wilson
Sandra Wityak
Yanyang Xiao
David Yun
Evelyn Zoernack

Tony Kennedy
Klaus Kusche
Larry Lambe
Franz Lehner
Howard Levy
Michael Lucks
Francois Maltey
Michael McGettrick
David Mentre
Mohammed Mobarak
Marc Moreno-Maza
Mark Murray
C. Andrew Neff
Arthur Norman
Summat Oemrawsingh
Julian A. Padget
Susan Pelzel
Ayal Pinkus
Gregorio Quintana-Orti
Norman Ramsey
Michael Richardson
Renaud Rioboo
Simon Robinson
Martin Rubey
William Schelter
Marshall Schor
Steven Segletes
William Sit
Fabio Stumbo
Moss E. Sweedler
T. Doug Telford
Mike Thomas
Barry Trager
Bernhard Wall
Juergen Weiss
James Wen
R. Clint Whaley
Berhard Will
Shmuel Winograd
Waldemar Wiwianka
Liu Xiaojun
Vadim Zhytnikov
Bruno Zuercher

Contents

New Foreword

On October 1, 2001 Axiom was withdrawn from the market and ended life as a commercial product. On September 3, 2002 Axiom was released under the Modified BSD license, including this document. On August 27, 2003 Axiom was released as free and open source software available for download from the Free Software Foundation's website, Savannah.

Work on Axiom has had the generous support of the Center for Algorithms and Interactive Scientific Computation (CAISS) at City College of New York. Special thanks go to Dr. Gilbert Baumslag for his support of the long term goal.

The online version of this documentation is roughly 1000 pages. In order to make printed versions we've broken it up into three volumes. The first volume is tutorial in nature. The second volume is for programmers. The third volume is reference material. We've also added a fourth volume for developers. All of these changes represent an experiment in print-on-demand delivery of documentation. Time will tell whether the experiment succeeded.

Axiom has been in existence for over thirty years. It is estimated to contain about three hundred man-years of research and has, as of September 3, 2003, 143 people listed in the credits. All of these people have contributed directly or indirectly to making Axiom available. Axiom is being passed to the next generation. I'm looking forward to future milestones.

With that in mind I've introduced the theme of the "30 year horizon". We must invent the tools that support the Computational Mathematician working 30 years from now. How will research be done when every bit of mathematical knowledge is online and instantly available? What happens when we scale Axiom by a factor of 100, giving us 1.1 million domains? How can we integrate theory with code? How will we integrate theorems and proofs of the mathematics with space-time complexity proofs and running code? What visualization tools are needed? How do we support the conceptual structures and semantics of mathematics in effective ways? How do we support results from the sciences? How do we teach the next generation to be effective Computational Mathematicians?

The "30 year horizon" is much nearer than it appears.

Tim Daly
CAISS, City College of New York
November 10, 2003 ((iHy))

Ultimately we would like Axiom to be able to prove that an algorithm generates correct results. There are many steps between here and that goal, including proving one Axiom algorithm correct through all of the levels from Spad code, to the Lisp code, to the C code, to the machine code; a daunting task of its own.

The proof of a single Axiom algorithm is done with an eye toward automating the process. Automated machine proofs are not possible in general but will certainly exist for known algorithms. Bressoud said:

> **Writing is nature's way of letting you know how sloppy your thinking is** – Guindon[Lamp02]
>
> **Mathematics is nature's way of letting you know how sloppy your writing is.** – Leslie Lamport[Lamp02]
>
> **The existence of the computer is giving impetus to the discovery of algorithms that generate proofs. I can still hear the echos of the collective sigh of relief that greeted the announcement in 1970 that there is no general algorithm to test for integer solutions to polynomial Diophantine equations; Hilbert's tenth problem has no solution. Yet, as I look at my own field, I see that creating algorithms that generate proofs constitutes some of the most important mathematics being done. The all-purpose proof machine may be dead, but tightly targeted machines are thriving.** – Dave Bressoud [Bres93]
>
> **In contrast to humans, computers are good at performing formal processes. There are people working hard on the project of actually formalizing parts of mathematics by computer, with actual formally correct formal deductions. I think this is a very big but very worthwhile project, and I am confident that we will learn a lot from it. The process will help simplify and clarify mathematics. In not too many years, I expect that we will have interactive computer programs that can help people compile significant chunks of formally complete and correct mathematics (based on a few perhaps shaky but at least explicit assumptions) and that they will become part of the standard mathematicians's working environment.** – William P. Thurston [Thur94]
>
> **Our basic premise is that the ability to construct and modify programs will not improve without a new and comprehensive look at the entire programming process. Past theoretical research, say, in the logic of programs, has tended to focus on methods for reasoning about individual programs; little has been done, it seems to us, to develop a sound understanding of the process of programming – the process by which programs evolve in concept and in practice. At present, we lack the means to describe the techniques of program construction and improvement in ways that properly link verification, documentation and adaptability.**

– Scherlis and Scott (1983) in [Maso86]

Chapter 1

Here is a problem

The goal is to prove that Axiom's implementation of the Euclidean GCD algorithm is correct.

From category EuclideanDomain (EUCDOM) we find the implementation of the Euclidean GCD algorithm:

```
gcd(x,y) ==                --Euclidean Algorithm
   x:=unitCanonical x
   y:=unitCanonical y
   while not zero? y repeat
      (x,y):= (y,x rem y)
      y:=unitCanonical y   -- this doesn't affect the
                           -- correctness of Euclid's algorithm,
                           -- but
                           -- a) may improve performance
                           -- b) ensures gcd(x,y)=gcd(y,x)
                           --    if canonicalUnitNormal
   x
```

The unitCanonical function comes from the category IntegralDomain (INTDOM) where we find:

```
unitNormal: % -> Record(unit:%,canonical:%,associate:%)
    ++ unitNormal(x) tries to choose a canonical element
    ++ from the associate class of x.
    ++ The attribute canonicalUnitNormal, if asserted, means that
    ++ the "canonical" element is the same across all associates of x
    ++ if \spad{unitNormal(x) = [u,c,a]} then
    ++ \spad{u*c = x}, \spad{a*u = 1}.
unitCanonical: % -> %
    ++ \spad{unitCanonical(x)} returns \spad{unitNormal(x).canonical}.
```

implemented as

```
UCA ==> Record(unit:%,canonical:%,associate:%)
if not (% has Field) then
  unitNormal(x) == [1$%,x,1$%]$UCA -- the non-canonical definition
unitCanonical(x) == unitNormal(x).canonical -- always true
recip(x) == if zero? x then "failed" else _exquo(1$%,x)
unit?(x) == (recip x case "failed" => false; true)
if % has canonicalUnitNormal then
   associates?(x,y) ==
     (unitNormal x).canonical = (unitNormal y).canonical
 else
   associates?(x,y) ==
     zero? x => zero? y
     zero? y => false
     x exquo y case "failed" => false
     y exquo x case "failed" => false
     true
```

1.1 Approaches

There are several systems that could be applied to approach the proof.

The plan is to initially look at Coq and ACL2. Coq seems to be applicable at the Spad level. ACL2 seems to be applicable at the Lisp level. Both levels are necessary for a proper proof.

Coq is very close to Spad in spirit so we can use it for the high-level proofs.

ACL2 is a Lisp-level proof technology which can be used to prove the Spad-to-Lisp level.

There is an LLVM to ACL2 translator which can be used to move from the GCL Lisp level to the hardware since GCL compiles to C. In particular, the "Vellvm: Verifying the LLVM" [Zdan14] project is important.

Quoting from Hardin [Hard14]

> LLVM is a register-based intermediate in Static Single Assignment (SSA) form. As such, LLVM supports any number of registers, each of which is only assigned once, statically (dynamically, of course, a given register can be assigned any number of times). Appel has observed that "SSA form is a kind of functional programming"; this observation, in turn, inspired us to build a translator from LLVM to the applicative subset of Common Lisp accepted by the ACL2 theorem prover. Our translator produces an executable ACL2 specification that is able to efficiently support validation via testing, as the generated ACL2 code features tail recursion, as well as in-place updates via ACL2's single-threaded object (stobj) mechanism. In order to ease the process of proving properties about these translated functions, we have also developed a technique for reasoning about tail-recursive ACL2 functions that execute in-place, utilizing a formally proven "bridge" to primitive-recursive versions of those functions operating on lists.

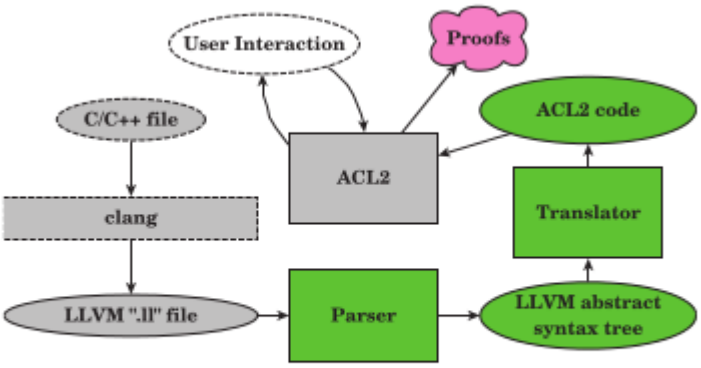

Figure 1: LLVM-to-ACL2 translation toolchain.

Hardin [Hard13] describes the toolchain thus:

> Our translation toolchain architecture is shown in Figure 1. The left side of tthe figure depicts a typical compiler frontend producing LLVM intermediate code. LLVM output can be produced either as a binary "bitcode" (.bc) file, or as text (.ll file). We chose to parse the text form, producing an abstract syntax tree (AST) representation of the LLVM program. Our translator then converts the AST to ACL2 source. The ACL2 source file can then be admitted into an ACL2 session, along with conjectures that one wishes to prove about the code, which ACL2 processes mostly automatically. In addition to proving theorems about the translated LLVM code, ACL2 can also be used to execute test vectors at reasonable speed.

Note that you can see the intermediate form from clang with

```
clang -O4 -S -emit-llvm foo.c
```

Both Coq and the Hardin translator use OCAML [OCAM14] so we will have to learn that language.

Chapter 2

Theory

The proof of the Euclidean algorithm has been known since Euclid. We need to study an existing proof and use it to guide our use of Coq along the same lines, if possible. Some of the "obvious" natural language statements may require Coq lemmas.

From WikiProof [Wiki14a] we quote:

Let

$$a, b \in \mathbf{Z}$$

and $a \neq 0 or b \neq 0$.

The steps of the algorithm are:

1. Start with (a, b) such that $|a| \geq |b|$. If $b = 0$ then the task is complete and the GCD is a.
2. if $b \neq 0$ then you take the remainder r of a/b.
3. set $a \leftarrow b$, $b \leftarrow r$ (and thus $|a| \geq |b|$ again).
4. repeat these steps until $b = 0$

Thus the GCD of a and b is the value of the variable a at the end of the algorithm.

The proof is:

Suppose

$$a, b \in \mathbf{Z}$$

and $a or b \neq 0$.

From the **division theorem**, $a = qb + r$ where $0 \leq r \leq |b|$

From **GCD with Remainder**, the GCD of a and b is also the GCD of b and r.

Therefore we may search instead for the $gcd(b, r)$.

Since $|r| \geq |b|$ and

$$b \in \mathbf{Z}$$

, we will reach $r = 0$ after finitely many steps.

At this point, $gcd(r, 0) = r$ from **GCD with Zero**.

We quote the **Division Theorem** proof [Wiki14b]:

For every pair of integers a, b where $b \neq 0$, there exist unique integers q, r such that $a = qb+r$ and $0 \leq r \leq |b|$.

Chapter 3

Software Details

3.1 Installed Software

Install CLANG, LLVM

```
http://llvm.org/releases/download.html
```

Install OCAML

```
sudo apt-get install ocaml
```

An OCAML version of gcd would be written

```
let rec gcd a b = if b = 0 then a else gcd b (a mod b)
val gcd : int -> int -> int = <fun>
```

Leslie Lamport[Lamp14] on 21^{st} Century Proofs.

A method of writing proofs is described that makes it harder to prove things that are not true. The method, based on hierarchical structuring, is simple and practical. The author's twenty years of experience writing such proofs is discussed.

Lamport points out that proofs need rigor and precision. Structure and Naming are important. Every step of the proof names the facts it uses.

Temporal Logic of Actions (TLA)

Sloppiness is easier than precision and rigor – Leslie Lamport[Lamp14a]

Computerising Mathematical Text[Kama15] explores various ways of capturing mathematical reasoning.

Chlipala[Chli15] gives a pragmatic approach to COQ.

Medina-Bulo et al.[Bulo04] gives a formal verification of Buchberger's algorithm using ACL2 and Common Lisp.

Théry[Ther01] used COQ to check an implementation of Buchberger's algorithm.

Pierce[Pier15] has a Software Foundations course in COQ with downloaded files in Pier15.tgz.

Chapter 4

Bibliography

Bibliography

[Bres93] David Bressoud. Review of the problems of mathematics. *Math. Intell.*, 15(4):71–73, 1993.

[Bulo04] I. Medina-Bulo, F. Palomo-Lozano, J.A. Alonso-Jiménez, and J.L. Ruiz-Reina. Verified computer algebra in acl2. *ASIC 2004, LNAI 3249*, pages 171–184, 2004.

> **Abstract:** In this paper, we present the formal verification of a Common Lisp implementation of Buchberger's algorithm for computing Groebner bases of polynomial ideals. This work is carried out in the ACL2 system and shows how verified Computer Algebra can be achieved in an executable logic.

[Chli15] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2015.

[Hard13] David S. Hardin, Jedidiah R. McClurg, and Jennifer A. Davis. Creating formally verified components for layered assurance with an llvm to acl2 translator.

> **Abstract:** This paper describes an effort to create a library of formally verified software component models from code that have been compiled using the Low-Level Virtual Machine (LLVM) intermediate form. The idea is to build a translator from LLVM to the applicative subset of Common Lisp accepted by the ACL2 theorem prover. They perform verification of the component model using ACL2's automated reasoning capabilities.

[Hard14] David S. Hardin, Jennifer A. Davis, David A. Greve, and Jedidiah R. McClurg. Development of a translator from llvm to acl2.

> **Abstract:** In our current work a library of formally verified software components is to be created, and assembled, using the Low-Level Virtual Machine (LLVM) intermediate form, into subsystems whose top-level assurance relies on the assurance of the individual components. We have thus undertaken a project to

build a translator from LLVM to the applicative subset of Common Lisp accepted by the ACL2 theorem prover. Our translator produces executable ACL2 formal models, allowing us to both prove theorems about the translated models as well as validate those models by testing. The resulting models can be translated and certified without user intervention, even for code with loops, thanks to the use of the def::ung macro which allows us to defer the question of termination. Initial measurements of concrete execution for translated LLVM functions indicate that performance is nearly 2.4 million LLVM instructions per second on a typical laptop computer. In this paper we overview the translation process and illustrate the translator's capabilities by way of a concrete example, including both a functional correctness theorem as well as a validation test for that example.

[Kama15] Fairouz Kamareddine, Joe Wells, Christoph Zengler, and Henk Barendregt. Computerising mathematical text, 2015.

Abstract: Mathematical texts can be computerised in many ways that capture differing amounts of the mathematical meaning. At one end, there is document imaging, which captures the arrangement of black marks on paper, while at the other end there are proof assistants (e.g. Mizar, Isabelle, Coq, etc.), which capture the full mathematical meaning and have proofs expressed in a formal foundation of mathematics. In between, there are computer typesetting systems (e.g. Latex and Presentation MathML) and semantically oriented systems (e.g. Content MathML, OpenMath, OMDoc, etc.). In this paper we advocate a style of computerisation of mathematical texts which is flexible enough to connect the diferent approaches to computerisation, which allows various degrees of formalsation, and which is compatible with different logical frameworks (e.g. set theory, category theory, type theory, etc.) and proof systems. The basic idea is to allow a man-machine collaboration which weaves human input with machine computation at every step in the way. We propose that the huge step from informal mathematics to fully formalised mathematics be divided into smaller steps, each of which is a fully developed method in which human input is minimal.

[Lamp02] Leslie Lamport. *Specifying Systems*. Addison-Wesley, 2002.

[Lamp14] Leslie Lamport. How to write a 21^{st} century proof, 2014.

Abstract: A method of writing proofs is described that makes it harder to prove things that are not true. The method, based on hierarchical structuring, is simple and practical. The author's twenty years of experience writing such proofs is discussed.

[Lamp14a] Leslie Lamport. Talk: How to write a 21^{st} century proof, 2014.

Comment: 2nd Heidelberg Laureate Forum Lecture Tuesday Sep 23, 2014

[Maso86] Ian A. Mason. *The Semantics of Destructive Lisp*. Center for the Study of Language and Information, 1986.

Abstract: Our basic premise is that the ability to construct and modify programs will not improve without a new and comprehensive look at the entire programming process. Past theoretical research, say, in the logic of programs, has tended to focus on methods for reasoning about individual programs; little has been done, it seems to us, to develop a sound understanding of the process of programming – the process by which programs evolve in concept and in practice. At present, we lack the means to describe the techniques of program construction and improvement in ways that properly link verification, documentation and adaptability.

[OCAM14] unknown. The ocaml website.

[Pier15] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catalin Hritcu, Vilhelm Sjoberg, and Brent Yorgey. Software foundations, 2015.

Abstract: This electronic book is a course on Software Foundations, the mathematical underpinnings of reliable software. Topics include basic concepts of logic, computer-assisted theorem proving, the Coq proof assistant, functional programming, operational semantics, Hoare logic, and static type systems. The exposition is intended for a broad range of readers, from advanced undergraduates to PhD students and researchers. No specific background in logic or programming languages is assumed, though a degree of mathematical maturity will be helpful. The principal novelty of the course is that it is one hundred per cent formalized and machine-checked: the entire text is literally a script for Coq. It is intended to be read alongside an interactive session with Coq. All the details in the text are fully formalized in Coq, and the exercises are designed to be worked using Coq. The files are organized into a sequence of core chapters, covering about one semester's worth of material and organized into a coherent linear narrative, plus a number of appendices covering additional topics. All the core chapters are suitable for both upper-level undergraduate and graduate students.

[Ther01] Laurent Théry. A machine-checked implementation of buchberger's algorithm. *Journal of Automated Reasoning*, 26:107–137, 2001.

Abstract: We present an implementation of Buchberger's algorithm that has been proved correct within the proof assistant Coq. The implementation contains the basic algorithm plus two standard optimizations.

[Thur94] William P. Thurston. On proof and progress in mathematics. *Bulletin AMS*, 30(2), April 1994.

[Wiki14a] ProofWiki. Euclidean algorithm.

[Wiki14b] ProofWiki. Division theorem.

[Zdan14] Steve Zdancewic and Milo M.K. Martin. Vellvm: Verifying the llvm.

Chapter 5

Index