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

April 15, 2018

27a6c3636e8a29e74cd7f58e4e93c30a0e05334e

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
Jerry Archibald	S.J. Atkins	Jeremy Avigad
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
Wolfgang Brehm	Peter A. Broadbery	Martin Brock
Manuel Bronstein	Christopher Brown	Stephen Buchwald
Florian Bundschuh	Luanne Burns	William Burge
Ralph Byers	Quentin Carpent	Pierre Casteran
Robert Cavines	Bruce Char	Ondrej Certik
Tzu-Yi Chen	Bobby Cheng	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
Karl Crary	Jeremy Du Croz	David Cyganski
Nathaniel Daly	Timothy Daly Sr.	Timothy Daly Jr.
James H. Davenport	David Day	James Demmel
Didier Deshommes	Michael Dewar	Inderjit Dhillon
Jack Dongarra	Jean Della Dora	Gabriel Dos Reis
Claire DiCrescendo	Sam Dooley	Nicolas James Doye
Zlatko Drmac	Lionel Ducos	Iain Duff
Lee Duhem	Martin Dunstan	Brian Dupee
Dominique Duval	Robert Edwards	Hans-Dieter Ehrich
Heow Eide-Goodman	Lars Erickson	Mark Fahey
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	Nicholas J. Higham	Hoon Hong
Roger House	Gernot Hueber	Pietro Iglio
Alejandro Jakubi	Richard Jenks	Bo Kagstrom
William Kahan	Kyriakos Kalorkoti	Kai Kaminski
Grant Keady	Wilfrid Kendall	Tony Kennedy
David Kincaid	Keshav Kini	Ted Kosan

Paul Kosinski	Igor Kozachenko	Fred Krogh
Klaus Kusche	Bernhard Kutzler	Tim Lahey
Larry Lambe	Kaj Laurson	Charles Lawson
George L. Legendre	Franz Lehner	Frederic Lehobey
Michel Levaud	Howard Levy	J. Lewis
Ren-Cang Li	Rudiger Loos	Craig Lucas
Michael Lucks	Richard Luczak	Camm Maguire
Francois Maltey	Osni Marques	Alasdair McAndrew
Bob McElrath	Michael McGettrick	Edi Meier
Ian Meikle	David Mentre	Victor S. Miller
Gerard Milmeister	Mohammed Mobarak	H. Michael Moeller
Michael Monagan	Marc Moreno-Maza	Scott Morrison
Joel Moses	Mark Murray	William Naylor
Patrice Naudin	C. Andrew Neff	John Nelder
Godfrey Nolan	Arthur Norman	Jinzhong Niu
Michael O'Connor	Summat Oemrawsingh	Kostas Oikonomou
Humberto Ortiz-Zuazaga	Julian A. Padget	Bill Page
David Parnas	Susan Pelzel	Michel Petitot
Didier Pinchon	Ayal Pinkus	Frederick H. Pitts
Frank Pfenning	Jose Alfredo Portes	E. Quintana-Orti
Gregorio Quintana-Orti	Beresford Parlett	A. Petitet
Andre Platzer	Peter Poromaas	Claude Quitte
Arthur C. Ralfs	Norman Ramsey	Anatoly Raportirenko
Guilherme Reis	Huan Ren	Albert D. Rich
Michael Richardson	Jason Riedy	Renaud Rioboo
Jean Rivlin	Nicolas Robidoux	Simon Robinson
Raymond Rogers	Michael Rothstein	Martin Rubey
Jeff Rutter	Philip Santas	David Saunders
Alfred Scheerhorn	William Schelter	Gerhard Schneider
Martin Schoenert	Marshall Schor	Frithjof Schulze
Fritz Schwarz	Steven Segletes	V. Sima
Nick Simicich	William Sit	Elena Smirnova
Jacob Nyffeler Smith	Matthieu Sozeau	Ken Stanley
Jonathan Steinbach	Fabio Stumbo	Christine Sundaresan
Klaus Sutner	Robert Sutor	Moss E. Sweedler
Eugene Surowitz	Yong Kiam Tan	Max Tegmark
T. Doug Telford	James Thatcher	Laurent Thery
Balbir Thomas	Mike Thomas	Dylan Thurston
Francoise Tisseur	Steve Toleque	Raymond Toy
Barry Trager	Themos T. Tsikas	Gregory Vanuxem
Kresimir Veselic	Christof Voemel	Bernhard Wall
Stephen Watt	Andreas Weber	Jaap Weel
Juergen Weiss	M. Weller	Mark Wegman
James Wen	Thorsten Werther	Michael Wester
R. Clint Whaley	James T. Wheeler	John M. Wiley
Berhard Will	Clifton J. Williamson	Stephen Wilson
Shmuel Winograd	Robert Wisbauer	Sandra Wityak
Waldemar Wiwianka	Knut Wolf	Yanyang Xiao
Liu Xiaojun	Clifford Yapp	David Yun
Qian Yun	Vadim Zhytnikov	Richard Zippel
Evelyn Zoernack	Bruno Zuercher	Dan Zwillinger

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 exist for known algorithms.

> **Q: Why bother doing proofs about programming languages? They are almost always boring if the definitions are right.**
>
> **A: The definitions are almost always wrong.**
>
> **If programming is understood not as the writing of instructions for this or that computing machine but as the design of methods of computation that it is the computer's duty to execute, then it no longer seems possible to distinguish the discipline of programming from constructive mathematics.**
> – Per Martin-Löf [Mart79]
>
> **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]
>
> **The intrinsically discrete nature of symbol processing makes programming such a tricky job that the application of formal techniques becomes a necessity.**
> – Edsger W. Dijkstra [Dijk83]
>
> **The notion of a proof serves not only to organize information, but to direct the analysis of a problem and produce the necessary insights. It is as much an analytical tool as it is a final product.**
> – Bates and Constable [Bate85]
>
> **By June 1949 people had begun to realize that it was not so easy to get programs right as at one time appeared. ... the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.**
> – Maurice Wilkes [Wilk85a]
>
> **I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. If this is true, building software will always be hard. There is inherently no silver bullet.**
> – Fredrick P. Brooks Jr. [Broo87]

I hold the opinion that the construction of computer programs is a mathematical activity like the solution of differential equations, that programs can be derived from their specifications through mathematical insight, calculation, and proof, using algebraic laws as simple and elegant as those of elementary arithmetic.
– C. A. R. Hoare [Fetz88]

It might be said that programs are conjectures, while executions are attempted refutations.
– James Fetzer [Fetz88]

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 mathematician's working environment.
– William P. Thurston [Thur94]

...constructive mathematics provides a way of viewing the language of logical propositions as a *specification* language for programs. An ongoing thrust of work in computer science has been to develop program specification languages and formalisms for systematically deriving programs from specifications. For constructive mathematics to provide such a methodology, techniques are needed for systematically extracting programs from constructive proofs. Early work in this field includes that of Bishop and Constable [Cons98]. What distinguished Martin-Löf's '82 type theory was that the method it suggested for program synthesis was exceptionally simple: a direct correspondence was set up between the constructs of mathematical logic, and the constructs of a functional programming language. Specifically, every proposition was considered to be isomorphic to a type expression, and the proof of a proposition would suggest precisely how to construct an inhabitant of the type, which would be a term in a functional programming language. The term that inhabits the type corresponding to a proposition is often referred to as the *computational content* of the proposition.

– Paul Bernard Jackson [Jack95]

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]

Type theory is nothing short of a grand unified theory of computation unified with mathematics so ultimately there is no difference between math and the code.
– Robert Harper [Harp13]

Informal proofs are algorithms. Formal proofs are code.
– Benjamin Pierce [Pier17]

Chapter 1

Why this effort will not succeed

This is an effort to prove Axiom correct. That is, we wish to prove that the algorithms provide correct, trustworthy answers.

All prior attempts at combinining a Computer Algebra system and a Proof system raise the issue that the CAS is untrustworthy.

Axiom tries to encode mathematical algorithms. Unlike other systems it is built on the scaffold of group theory which provides a sound mathematical foundation. As such, it seems only reasonable that the algorithms in Axiom can be proven correct, hence the project to Prove Axiom Correct (PAC).

The PAC project will not succeed. This is perfectly obvious from the outset. But, given the law of the excluded middle (that is, $A \vee \ A$) is not applicable in this case, the fact that the project "does not succeed" does not imply failure. Learning will occur.

That said, we can list quite a few reasons why PAC will not succeed, most of which are explained in more detail in [Cyph17]. We provide useful names for the likely criticisms presented there and paraphrase those criticisms applied to our context.

1. **Leap of Faith** All of these efforts either require making a leap of faith to go from verified code to a real-world problem, or required the use of an artificially restricted system in order to function (the NewSpeak approach, create a language in which it is impossible to think bad thoughts), indicating that formal verification down to the binary code level is unlikely to be practical in any generally accepted formal methods sense.
2. **Tools for Toy Problems** The tools used to support formal methods arose from an academic research envvironment characterised by a small number of highly skilled users. The tools were tested on small problems (usually referred to somewhat disparagingly as "toy problems") which were targeted more at exercising the tools than exercising the problem.
3. **Opaque Specifications** Another factor which complicates the use of formal methods is that the mathematical methods available to software engineers are often very difficult to use and plagued by notation which is cumbersome and hard to read and understand.
4. **Too Large a Task** The national technology base for this level of task is essentially non-existent. There do not appear to be even 20 people in the world that have undertaken the essential steps. "In order to get a system with excellent system integrity, you must

ensure that it is designed and built by geniuses. Geniuses are in short supply" [Blak96].

5. **False Axioms** Problems occur when a proof has to be manually augmented with "self-vidient" truths which sometimes turn out to be false and end up misguiding the prover.
6. **Intellectual Cost** Requiring a proof of an algorithm adds to the already outsized cost of creating the algorithm in Axiom, given the steep intellectual hill such a system already presents.
7. **Ripple Cost** Multiple layers of abstraction are required to go from a proof to its implementation and a change in any of the layers can result in a ripple effect as changes propagate. If a change manages to propagate its way into a formally proven section, portions of, or possibly all of, the proof might need to be redone.
8. **Informal Implementation Details** Current techniques rarely reach down any further than the high-level specificaiton resulting in large amounts of time and energy being poured into producing a design specification which is void of any implementation details.
9. **Specification Mismatch** Formal proofs that a specification is correct don't show that the assumptions made in the proof are correct in the actual physical system. If the code which is supposed to implement the formal specification doesn't quite conform to it or the compiler can't produce an executable which quite matches what was intended in the code then no amount of formal proving will be able to guarantee the execution behaviour of the code.
10. **Natural / Formal Mismatch** Natural language descriptions of what is intended may mismatch the formal language descriptions. In many cases "common sense" assumptions come into play which are never formally stated.
11. **Specification Flaws** The code may implement a specification exactly as stated but the specification can be flawed.
12. **Mis-implemented Specifications** The specification may be correct but the implementation does not match the specification.
13. **Specification Narrowing** The specification language may not be sufficient to state all of the required concepts so a "narrowing" of the specification is made to approximate the intent.
14. **Specification Widening** The specification language may generalize the required concepts so a "widening" of the specification is made beyond the required intent.
15. **Specification Impedence Mismatch** The specification language does not cover the concepts in the domain of interest, requiring considerable effort to "model" the domain.
16. **Specification Blindness** The specification writers can't take into account every eventuality which will arise. As a result, an implemention of a specification doesn't just implement it, it alters it in order to fit real-world constraints which weren't foreseen by the original authors of the specification.
17. **Contradictory Proofs** If the system is inconsistent, it would be possible to find two contradictory proofs. It is likely that such an inconsistency will not be detected and only one of the proofs will be accepted since the other one is never generated.
18. **Likely to be Ignored** There are thousands of proofs published, most of which are "write only" and are never checked. This is likely to be the case with Axiom's proofs.

19. **Proven Programs are Wrong** It is possible to prove a program correct and still get wrong results. Testing is still required.
20. **Proofs are a Social Process** Axiom's "social circle" is vanishingly small.
21. **Chicken and Egg** Do proofs follow from programs or programs follow from proofs?
22. **Boiling the Ocean** The task is much too large.
23. **MetaTheory Cost** There is a cost to developing theories for new data types; these are unnecessary "meta" costs.
24. **Partial Functions** Not all of the functions are total so it is difficult to prove theorems about them.
25. **Undecidable Theories** Some theories are known to be undecidable so there will be problems with their proofs.

Fenton [Fent93] states that there is no hard evidence to show that

1. formal methods have been used cost effectively on a real safety-critical system development
2. the use of formal methods can deliver reliability more cost effectively than traditional structured methods with enhanced testing
3. sufficient numbers of either developers or users can ever be trained to make proper use of formal methods

Calude, et al. [Calu07] take a quite different attack on the idea of proving and programming, specifically

1. Theorems (in mathematics) correspond to algorithms and not programs (in computer science); algorithms are subject to mathematical proofs (for example correctness)
2. The role of proof in mathematical modeling is very small: adequacy is the main issue
3. Programs (in computer science) correspond to mathematical models. They are not subject to proofs, but to an adequacy and relevance analysis; in this type of analysis, some proofs may appear. Correctness proofs in computer science (if any) are not cost-effective.
4. Rigour in programming is superior to rigour in mathematical proofs.
5. Programming gives mathematics a new form of understanding
6. Although the Hilbertian notion of proof has few chances to change, future proofs will be of various types and will play different roles, and their truth will be checked differently.
7. In general, correctness is undecidable
8. for most non-trivial cases, correctness is a monumental task which gives an added confidence at a disproportionate cost.

Still another attack comes from Hoare [Hoar96] which states that "By surveying current software engineering practice, this paper reveals that the techniques employed to achieve reliability are little different from those which have proved effective in all other branches of modern engineering: rigorous management of procedures for design inspection and review; quality assurance based on a wide range of targeted tests; continuous evolution by removal of errors from products already in widespread use; and defensive programming, among other forms of deliberate over-engineering. Formal methods and proof play a small direct role in large scale programming; but they do provide a conceptual framework and basic understanding to promote the best of current practice, and point directions for future improvement."

Fetzer [Fetz88] adds "The notion of program verification appears to trade upon an equvocation. Algorithms, as logical structures, are appropriate subjects for deductive verification. Programs, as causal models of those structures, are not. The success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility."

An unnamed letter writer to the CACM says "It is time somebody said it – loud and clear – the formal approach to software verification does not work now and probably never will." [Glas02]

DeMillo et al. [Demi79] says "It is argued that formal verifications of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics. Furthermore the absence of continuity, the inevitability of change, and the complexity of specification of significantly many real programs make the formal verification process difficult to justify and manage. It is felt that ease of formal verification should not dominate program language design.",

DeMillo argues that proofs are social constructs. "Mathematiical proofs increase our confidence in the truth of mathematical statements only after they have been subjected to the social mechanisms of the mathematical community. These same mechanisms doom the so-called proofs of software, the long formal verifications that correspond, not to the working mathematical proof, but to the imaginary logical structure that the mathematician conjures up to describe his feeling of belief. Verifications are not messages, a person who ran out into the hall to communicate his latest verification would rapidly find himself a social pariah. Verifications cannot really be read; a reader can flay himself through one of the shorter ones by dint of heroic effort, but that's not reading. Being unreadable and – literally – unspeakable, verifications cannot be internalized, transformed, generalized, used, connected to other disciplines, and eventually incorporated into a community consciousness. They cannot acquire credibility gradually, as a mathematical theorem does; one either believes them blindly, as a pure act of faith, or not at all."

And still more with "There is a fundamental logical objection to verification, an objection on its own ground of formalistic rigor. Since the requirement for a program is informal and the program is formal, there must be a transition, and the transition itself must necessarily be informal."

And "So, having for the moment suspended all rational disbelief, let us suppose that the programmer gets the message "VERIFIED." And let us suppose further that the message does not result from a failure on the part of the verifying system. What does the programmer know? He knows that his program is formally, logically, provably, certifiably correct. He does not know, however, to what extent it is reliable, dependable, trustworthy, safe; he does not know within what limits it will work; he does not know what happens when it exceeds those limits. And yet he has that mystical stamp of approval "VERIFIED.""

Manna and Waldinger [Mann78] mentions

1. We can never be sure that the specifications are correct
2. No verification system can verify every correct program
3. We can never be certain that a verification system is correct

Hall [Hall90] presents 7 myths of formal methods.

1. Formal methods can guarantee that software is perfect.

2. They work by proving that programs are correct.
3. Only highly critical systems benefit from their use.
4. They involve complex mathematics.
5. They increase the cost of development.
6. They are incomprehensible to clients.
7. Nobody uses them for real projects.

Bowen and Hinchey [Bowe95] follow up with 7 more myths.

1. Formal methods delay the developent process
2. Formal methods lack tools
3. Formal methods replace traditional engineering design methods
4. Formal methods only apply to software
5. Formal methods are unnecessary
6. Formal methods are not supported
7. Formal methods people always use formal methods

Chapter 2

Progress Will Occur

At the very lowest level there have been some truly impressive steps toward formal verification of low-level implmentation details.

Reid [Reid17] has created and verified a huge specification (over 1/2 Million nodes) of the ARM processor. The specification passes almost every test in their huge (11k) test suite. Using the specification with an SMT Solver allows the ability to do things like ask what input will give the known output, for example.

Chlipala has done both machine-level [Chli17a] and higher-level [Chli17] formal proofs. He claims that within 10 years it will be normal to have deployed computer systems with top-to-bottom correctness proofs which were checked algorithmically.

Chapter 3

Here is a problem

Axiom has a domain-specific language for mathematical computation called Spad. This is implemented in Common Lisp. Our version of Common Lisp compiles to C. The C code is then compiled to machine code. The machine code is executed by a physical machine.

There are various attacks on each of these areas. We will be addressing them all at some point, from the mathematical specification, through the Spad code all the way down to the hardware timing signals.

One useful technique to exploit is mentioned by Myreen, Slind, and Gordon [Myre09a]. They 'forward compile' the code to f, collect the result as f', and then prove the $f = f'$. This proof is sufficient to show that the compile step is correct.

3.1 Proving the Algebra

3.1.1 Defining the Spad syntax

3.1.2 Defining the Spad semantics

Floyd [Floy67], Back [Back81], Caldwell [Cald97], Filliatre [Fill03], Filliatre and Paskevich [Fill13]

3.2 Proving the Logic

Proving programs correct involves working with a second programming language, the proof language, that is well-founded on some theory. Proofs (programs), can be reduced (compiled) in this new language to the primitive constructs (machine language).

The ideal case would be that the programming language used, such as Spad, can be isomorphic, or better yet, syntactically the same as the proof language. Unfortunately that is not (yet?) the case with Spad.

The COQ system language, Gallina, is the closest match to Spad.

Davis [Davi09], Myreen and Davis [Myre11], Myreen and Davis [Myre14], Davis and Myreen

[Davi15]

3.2.1 Defining the Algebra specifications

Talpin and Jouvelot [Talp92]

3.3 Proving the Lisp

Myreen [Myre12]

3.4 Proving the Compiler

Myreen [Myre10]

3.5 Proving to the metal

Myreen and Gordon [Myre09], Myreen, Slind, and Gordon [Myre09a]

Domipheus [Domi18] provides a VHDL (hardware description language) CPU which provides the actual electrical timing of signals all the way up to a machine language which consists of

OPERATION	FUNCTION
Add	D = A + B
Substrct	D = A - B
Bitwise Or	D = A or B
Bitwise Xor	D = A xor B
Bitwise And	D = A and B
Bitwise Not	B = not A
Read	D = Memory[A]
Write	Memory[A] = B
Load	D = 8-bit immediate value
Compare	D = cmp(A,B)
Shift Left	D = A << B
Shift Right	D = A >> B
Jump/Branch	PC = A Register or Immediate Value
Jump/Brance conditionally	PC = Register if (condition) else nop

3.6 Setting up the problem

The GCD function will be our first example of a proof.

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

We need to be clear about what is to be proven. In this case, we need to show that, given GCD(a,b),

1. **GCD** is a function from $a \times b \Rightarrow c$

2. **a** and **b** are elements of the correct type
3. **c**, the result, is the correct type
4. the meaning of **divisor**
5. the meaning of a **common divisor**
6. **GCD** terminates

We next need to set up the things we know in "the global environment", generally referred to as **E** in Coq.

Axiom's GCD is categorically defined to work over any Euclidean domain. This means that the axioms of a Euclidean domain are globally available. In fact, this is stronger than we need since

- commutative rings $\subset$ integral domains
- integral domains $\subset$ integrally closed domains
- integrally closed domains $\subset$ GCD domains
- GCD domains $\subset$ unique factorization domains
- unique factorization domains $\subset$ principal ideal domains
- principal ideal domains $\subset$ Euclidean domains

A Euclidean function on R is a function f from $\mathbb{R}$
$\{0\}$ to the non-negative integers satisfying the following fundamental division-with-remainder property [WikiED]:

$D(a, b) =$ set of common divisors of a and b.

$\gcd(a, b) = \max D(a, b)$

3.7 Axiom NNI GCD

NonNegativeInteger inherits `gcd` from `Integer` up the "add chain" since it is a subtype of `Integer`. `Integer` has `EuclideanDomain` as an ancestor [Book103]:

```
(1) -> getAncestors "Integer"

   (1)
   {AbelianGroup, AbelianMonoid, AbelianSemiGroup, Algebra, BasicType,
    BiModule, CancellationAbelianMonoid, CharacteristicZero, CoercibleTo,
    CombinatorialFunctionCategory, CommutativeRing, ConvertibleTo,
    DifferentialRing, EntireRing, EuclideanDomain, GcdDomain,
    IntegerNumberSystem, IntegralDomain, LeftModule, LeftOreRing,
    LinearlyExplicitRingOver, Module, Monoid, OpenMath, OrderedAbelianGroup,
    OrderedAbelianMonoid, OrderedAbelianSemiGroup,
    OrderedCancellationAbelianMonoid, OrderedIntegralDomain, OrderedRing,
    OrderedSet, PatternMatchable, PrincipalIdealDomain, RealConstant,
    RetractableTo, RightModule, Ring, Rng, SemiGroup, SetCategory, StepThrough,
    UniqueFactorizationDomain}
                                                        Type: Set(Symbol)
```

From category `EuclideanDomain` (EUCDOM) we find the implementation of the Euclidean GCD algorithm [Book102]:

```
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
```

Coq proves the following GCD function:

```
Fixpoint gcd a b :=
  match a with
    | 0 => b
    | S a' => gcd (b mod (S a')) (S a')
  end.
```

This can be translated directly to working Spad code:

```
GCD(x:NNI,y:NNI):NNI ==
  zero? x => y
  GCD(y rem x,x)
```

with the test case results of:

```
(1) -> GCD(2415,945)
   Compiling function mygcd2 with type (NonNegativeInteger,
      NonNegativeInteger) -> NonNegativeInteger

   (1)  105
                                                        Type: PositiveInteger
(2) -> GCD(0,945)

   (2)  945
                                                        Type: PositiveInteger
(3) -> GCD(2415,0)

   (3)  2415
                                                        Type: PositiveInteger
(4) -> GCD(17,15)

   (4)  1
                                                        Type: PositiveInteger
```

3.8 Mathematics

From Buchberger [Buch97],

Define "divides"

$$t|a \iff \exists u(t \cdot u = a)$$

Define "greatest common divisor"

$$\mathrm{GCD}(a,b) = \forall t\ max(t|a \wedge t|b)$$

Theorem:

$$(t|a \wedge t|b) \iff t|(a-b) \wedge t|b$$

Euclid's Algorithm

$$a > b \Rightarrow \mathrm{GCD}(a,b) = \mathrm{GCD}(a-b,b)$$

By the definition of GCD we need to show that

$$\forall t\ max(t|a \wedge t|b) = \forall t\ max(t|(a-b) \wedge t|b)$$

Thus we need to show that

$$(t|a \wedge t|b) \iff (t|(a-b) \wedge t|b)$$

Let t be arbitrary but fixed and assume

$$(t|a \wedge t|b) \tag{3.1}$$

We have to show

$$t|(a-b) \tag{3.2}$$

and

$$t|b \tag{3.3}$$

Equation 3.3 follows propositionally. For equation 3.2, by definition of "divides", we have to find a w such that

$$t \cdot w = a - b \tag{3.4}$$

From 3.1, by definition of "divides", we know that for certain u and v

$$t \cdot u = a$$

and

$$t \cdot v - b$$

Hence,

$$a - b = t \cdot u - t \cdot v$$

But

$$t \cdot u - t \cdot v = t \cdot (u - v)$$

So we need to find

$$w = u - v$$

and

$$\text{Find w such that } t \cdot u - t \cdot v = t \cdot w$$

3.9 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.

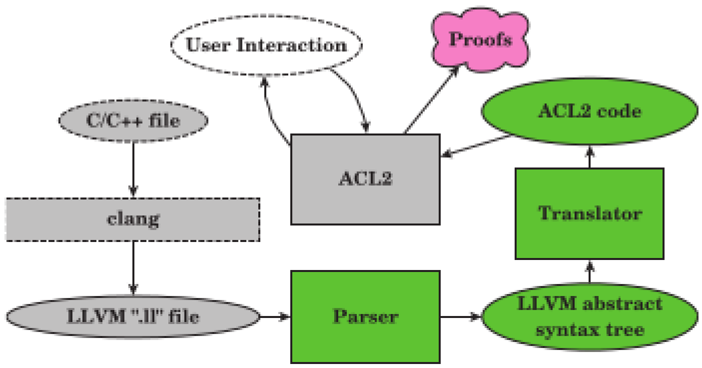


Figure 1: LLVM-to-ACL2 translation toolchain.

Hardin [Hard13] describes the toolchain thus:

> Our translation toolchain architecture is shown in Figure 1. The left side of the 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 4

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 $0 \leq r \leq |b|$.

4.0.1 Hoare's axioms and gcd proof

From Hoare [Hoar69]

A1	$x + y = y + x$	addition is commutative
A2	$x \times y = y \times x$	multiplication is commutative
A3	$(x + y) + z = x + (y + z)$	addition is associative
A4	$(x \times y) \times z = x \times (y \times z)$	multiplication is associative
A5	$x \times (y + z) = x \times y + x \times z$	multiplication distributes through addition
A6	$y \leq x \rightarrow (x - y) + y = x$	addition cancels subtraction
A7	$x + 0 = x$	
A8	$x \times 0 = 0$	
A9	$x \times 1 = x$	

D0 Axiom of Assignment

$$\vdash P_0\{x := f\}P$$

where

- x is a variable identifier
- f is an expression
- P_0 is obtained from P by substituting f for all occurrences of x

4.1 The Division Algorithm

From Judson [Juds15],

An Application of the Principle of Well-Ordering that we will use often is the division algorithm.

Theorem 2.9 Division Algorithm Let a and b be integers, with $b > 0$. Then there exists unique integers q and r such that

$$a = bq + r$$

where $0 \leq r < b$.

Proof

Let a and b be integers. If $b = ak$ for some integer k, we write $a|b$. An integer d is called a *common divisor* of a and b if $d|a$ and $d|b$. The *greatest common divisor* of integers a and b is a positive integer d such that d is a common divisor of a and b and if $d^{'}$ is any other common divisor of a and b, then $d^{'}|d$. We write $d = gcd(a, b)$; for example, $gcd(24, 36) = 12$ and $gcd(120, 102) = 6$. We say that two integers a and b are *relatively prime* if $gcd(a, b) = 1$.

Theorem 2.10 Let a and b be nonzero integers. Then there exist integers r and s such that

$$gcd(a, b) = ar + bs$$

Furthermore, the greatest common divisor of a and b is unique.

Proof

Corollary 2.11 Let a and b be two integers that are relatively prime. Then there exist integers r and s such that

$$ar + bs = 1$$

The Euclidean Algorithm

Among other things, Theorem 2.10 allows us to compute the greatest common divisor of two integers.

Example 2.1.2 Let us compute the greatest common divisor of 945 and 2415. First observe that

$$\begin{aligned} 2415 &= 945 \cdot 2 + 525 \\ 945 &= 525 \cdot 1 + 420 \\ 525 &= 420 \cdot 1 + 105 \\ 420 &= 105 \cdot 4 + 0 \end{aligned}$$

Reversing our steps, 105 divides 420, 105 divides 525, 105 divides 945, and 105 divides 2415. Hence, 105 divides both 945 and 2415. If d were another common divisor of 945 and 2415, then d would also have to divide 105. Therefore, $gcd(945, 2415) = 105$.

If we work backward through the above sequence of equations, we can also obtain numbers r and s such that

$$945r + 2415s = 105$$

$$\begin{aligned} 105 &= 525 + (-1) \cdot 420 \\ 105 &= 525 + (-1) \cdot [945 + (-1) \cdot 525] \\ 105 &= 2 \cdot 525 + (-1) \cdot 945 \\ 105 &= 2 \cdot [2415 + (-2) \cdot 945] + (-1) \cdot 945 \\ 105 &= 2 * 2415 + (-5) \cdot 945 \end{aligned}$$

So $r = -5$ and $s - 2$. Notice the r and s are not unique, since $r = 41$ and $s = -16$ would also work.

To compute $gcd(a, b) = d$, we are using repeated divisions to obtain a decreasing sequence of positive integers $r_1 > r_2 > \ldots > r_n = d$; that is

$$\begin{aligned} b &= aq_1 + r_1 \\ a &= r_1 q_2 + r_2 \\ r_1 &= r_2 q_3 + r_3 \\ &\vdots \\ r_{n-2} &= r_{n-1} q_n + r_n \\ r_{n-1} &= r_n q_{n+1} \end{aligned}$$

To find r and s such that $ar + bs = d$, we begin with the last equation and substitute results obtained from the previous equations:

$$\begin{aligned} d &= r_n \\ d &= r_{n-2} - r_{n-1} q_n \\ d &= r_{n-2} - q_n (r_{n-3} - q_{n-1} r_{n-2} \\ d &= -q_n r_{n-3} + (1 + q_n q_{n-1}) r_{n-2} \\ &\vdots \\ d &= ra + sb \end{aligned}$$

Chapter 5

GCD in Nuprl by Anne Trostle

Quoted from [Tros13]:

Here we show how to use the Nuprl proof assistant to develop an existence proof for the greatest common divisor of two natural numbers. We then take the proof a step further and show that the greatest common divisor, or GCD, can be calculated as a linear combination of the two numbers. For each proof, we also show that Nuprl can extract a *program* from the proof that can be used to perform calculations.

The greatest common divisor is defined in Nuprl as follows:

Defintion 1: gcd_p

$GCD(m : n : g) == (g|m) \wedge (g|n) \wedge (\forall z : Z.(((z|m) \wedge (z|m)) \rightarrow (z|g)))$

Defintion 2: divides

$b|a == \exists c : Z.(a = (b * c))$

In words, Definition 1 means that g is the greatest common divisor of m and n when g divides both m and n, and any other common divisor of m and n divides g.

To prove that the GCD exists, we are going to use Euclid's algorithm, whicc is based on the property that for two integers m and n, the GCD of m and n is equivalent to the GCD of n and the remainder from $m \div n$:

Lemma 1: div_rem_gcd_anne

$\forall m : Z. \quad \forall n : \mathbb{Z}^{-0}, \quad \forall g : \mathbb{Z}.(GCD(m; n; g) \iff GCD(n; m \text{ rem } n; g))$

Another useful fact about the GCD is that the GCD of an integer z and 0 is z. A proof of this property can be done by showing that each part of Definition 1 is satisfied.

Lemma 2: gcd_p_zero

$\forall z : \mathbb{Z}. \quad GCD(z; 0; z)$

From these properties we can see a method for calculating the greatest common divisor of two numbers: continue finding remainders until you reach 0 and then use the fact that the GCD of an integer z and 0 is z. Since the GCD stays the same as you reduce the terms, z is also the GCD of the original pair of numbers. This is Euclid's algorithm. Here is an example of how it works, using 18 and 12:

$$\begin{aligned}
\text{GCD(18;12;g)} &= \text{GCD(12;18 rem 12;g)} \\
&= \text{GCD(12;6;g} \\
&= \text{GCD(6;12 rem 6;g)} \\
&= \text{GCD(6;0;g)} \\
&\rightarrow \text{g} = 6
\end{aligned}$$

Using this idea we can not only prove that the GCD exists but we can also construct a method for actually computing the GCD. A great feature of Nuprl is that when we run a constructive existence proof, we can extract a program from it and use the program to perform calculations. In the next section we show in detail how to develop a constructive existence proof for the GCD using induction. Induction proofs often to hand-in-hand with recursive programs, and sure enough, a very clean recursive program can be extracted from the proof, and this program follows exactly the method we just came up with:

$\lambda n.letrcgcd(n) =$
$\lambda m.ifn = 0thenm$
$else(gcd(mremn)n)$
$ingcd(n)$

The program here is an example of *currying*: a function of n that results in another function which then uses m. This isn't necessarily intuitive, since when we think of the GCD we think of a function of a pair (or more) of numbers, so we might expect the program to start with something like "gcd$(m, n) = \ldots$". But the proof that follows uses natural induction on a single variable and flows very nicely, giving reason to prefer the curried function here. To develop a proof that produces a function of the pair (m, n) would require induction on the pair itself which isn't as intuitive or easy to understand as natural induction on a single variable.

Chapter 6

Software Details

6.1 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>
```

Chapter 7

Temporal Logic of Actions (TLA)

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

Leslie Lamport [Lamp14] [Lamp16] 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.

Quoting from [Lamp16]:

Broadly speaking, a TLA+ proof is a collection of *claims*, arranged in a hierarchical structure which we describe below, where each claim has an *assertion* that is either *unjustified* or justified by a collection of *cited facts*. The purpose of TLAPS is to check the user-provided proofs of theorems, that is, to check that the hierarchy of claims indeed establishes the truth of the theorem if the claims were true, and then to check that the assertion of every justified claim indeed is implied *by* its cited facts. If a TLA+ theorem has a proof with no unjustified claims, then, as a result of checking the proof, TLAPS verifies the truth of the theorem.

7.1 The algorithm

The well-known Euclidean algorithm can be written in the PlusCal language as follows:

```
--algorithm Euclid {
  variables x \in 1..M, y \in 1..N, x0 = x, y0 = y;
  {
    while (x # y) {
      if (x < y) { y := y - x; }
      else { x := x-y; }
    };
    assert x = GCD(x0, y0) /\ y = GCD(x0, y0)
  }
```

The PlusCal translator translates this algorithm into a TLA+ specification that we could prove correct. However, in this tutorial, we shall write a somewhat simpler specification of Euclid's algorithm directly in TLA+.

7.1.1 Creating a new TLA+ module

In order to get the definitions of arithmetic operators (+, −, etc.), we shall make this specification *extend* the `Integers` standard module.

```
--------------------- Module Euclid ----------------------
EXTENDS Integers
```

7.1.2 Definitions

We shall then define the GCD of two integers. For that purpose, let us define the predicate "p divides q" as follows: p divides q iff there exists some integer d in the interval 1..q such that q is equal to p times d.

```
p | q == \E d \in 1..q : q = p * d
```

We then define the set of divisors of an integer q as the sets of integers which both belong to the interval 1..q and divide q:

```
Divisors(q) == {d \in 1..q : d | q}
```

We define the maximum of a set S as one of the elements of this set which is greater than or equal to all the other elements:

```
Maximum(S) == CHOOSE x \in S : \A y \in S : x >= y
```

And finally, we define the GCD of two integers p and q to be the maximum of the intersection of Divisors(p) and Divisors(a):

```
GCD(p,q) == Maximum(Divisors(p) \cap Divisors(q))
```

For convenience, we shall also define the set of all positive integers as:

```
Number = Nat \ {0}
```

7.1.3 Constants and variables

We then define the two constants and two variables needed to describe the Euclidean algorithm, where M and N are the values whose GCD is to be computed:

```
CONSTANTS M, N
VARIABLES x, y
```

7.1.4 The specification

We define the initial state of the Euclidean algorithm as follows:

```
Init == (x = M) /\ (y = N)
```

In the Euclidean algorithm, two actions can be performed:

- set the value of y to y - x if x < y

- set the value of x to x - y if x > y

These actions are again written as a definition of `Next`, which specifies the next-state relation. In TLA+, a primed variable refers to its value at the next state of the algorithm.

```
Next == \/ /\ x < y
           /\ y' = y - x
           /\ x' = x
        \/ /\ y < x
           /\ x' = x-y
           /\ y' = y
```

The specification of the algorithm asserts that the variables have the correct initial values and, in each execution step, either a `Next` action is performed or x and y keep the same values:

```
Spec == Init /\ [][Next]_<<x,y>>
```

(For reasons that are irrelevant to this algorithm, TLA specifications always allow *stuttering steps* that leave all the variables unchanged.)

We want to prove that the algorithm always satisfies the following property:

```
ResultCorrect == (x = y) => x = GCD(M, N)
```

Hence we want to prove the following theorem named Correctness:

```
THEOREM Correctness == Spec => []ResultCorrect
```

7.1.5 Summary

```
--------------------- Module Euclid ----------------------
EXTENDS Integers

p | q == \E d \in 1..q : q = p * d
Divisors(q) == {d \in 1..q : d | q}
Maximum(S) == CHOOSE x \in S : \A y \in S : x >= y
GCD(p,q) == Maximum(Divisors(p) \cap Divisors(q))
Number == Nat \ {0}

CONSTANTS M, N
VARIABLES x, y

Init == (x = M) /\ (y = N)

Next == \/ /\ x < y
           /\ y' = y - x
           /\ x' = x
        \/ /\ y < x
           /\ x' = x-y
           /\ y' = y

Spec == Init /\ [][Next]_<<x,y>>

ResultCorrect == (x = y) => x = GCD(M,N)

THEOREM Correctness == Spec => []ResultCorrect
```

7.2 A simple proof

7.2.1 The invariant

Intuitively, the theorem Correctness holds because the implementation guarantees the following *invariant*

```
InductiveInvariant == /\ x \in Number
                      /\ y \in Number
                      /\ GCD(x, y) = GCD(M, N)
```

That is, `InductiveInvariant` holds for the initial state (i.e., the state specified by `Init`) and is preserved by the next-state relation `[Next]`$_{<<\, x, y\, >>}$

7.2.2 Checking proofs

First we need to assume that constants M and N are not equal to zero

```
ASSUME NumberAssumption == M \in Number /\ N \in Number
```

Let us then prove that `InductiveInvariant` holds for the initial state.

```
THEOREM InitProperty == Init => InductiveInvariant
```

To check whether TLAPS can prove that theorem by itself, we declare its proof obvious.

```
THEOREM InitProperty == Init => InductiveInvariant
  OBVIOUS
```

We now ask TLAPS to prove that theorem. But TLAPS does not know how to prove the proof obligation corresponding to that proof. It prints that obligation and reports failures to three backends, Zenon, Isabelle, and SMT. The default behavior of TLAPS is to send obligations first to an SMT solver (by default CVC3), then if that fails to the automatic prover Zenon, then if Zenon fails to Isabelle (with the tactic "auto").

7.2.3 Using facts and definitions

The obligation cannot be proved because TLAPS treats the symbols `Init` and `InductiveInvariant` as opaque identifiers unless it is explicitly instructed to expand their definitions using the directive `DEF`. The main purpose of this treatment of definitions is to make proof-checking tractable, because expanding definitions can arbitrarily increase the size of expressions. Explicit use of definitions is also a good hint to the (human) reader to look only at the listed definitions to understand a proof step. In that precise case, we can ask TLAPS to expand definitions of `Init` and `InductiveInvariant`, by replacing the proof `OBVIOUS` by the proof `BY DEF Init, InductiveInvariant`. In the obligations sent to the backends, the definitions of `Init` and `InductiveInvariant` have been expanded.

Unfortunately, none of the back-ends could prove that obligation. As with `definitions`, we have to specify which facts are *usable*. In this case, we have to make the fact `NumberAssumption` usable by changing the proof to

```
THEOREM InitProperty == Init => InductiveInvariant
  BY NumberAssumption DEF Init, InductiveInvariant
```

The general form of a `BY` proof is:

$$\texttt{BY}\ e_1, \ldots, e_m\ \texttt{DEF}\ d_1, \ldots, d_n$$

which claims that the assertion follows by assuming $e_1, \ldots, e_m$ and expanding the definitions $d_1, \ldots, d_n$. It is the job of TLAPS to then check this claim, and also to check that the cited facts $e_1, \ldots, e_m$ are indeed true.

Finally, SMT succeeds in proving that obligation.

```
--------------------- Module Euclid ----------------------
EXTENDS Integers

p | q == \E d \in 1..q : q = p * d
Divisors(q) == {d \in 1..q : d | q}
Maximum(S) == CHOOSE x \in S : \A y \in S : x >= y
GCD(p,q) == Maximum(Divisors(p) \cap Divisors(q))
Number == Nat \ {0}

CONSTANTS M, N
VARIABLES x, y

Init == (x = M) /\ (y = N)

Next == \/ /\ x < y
           /\ y' = y - x
           /\ x' = x
        \/ /\ y < x
           /\ x' = x-y
           /\ y' = y

Spec == Init /\ [][Next]_<<x,y>>

ResultCorrect == (x = y) => x = GCD(M,N)

InductiveInvariant == /\ x \in Number
                      /\ y \in Number
                      /\ GCD(x, y) = GCD(M, N)

ASSUME NumberAssumption == M \in Number /\ N \in Number

THEOREM InitProperty == Init => InductiveInvariant
  BY NumberAssumption DEF Init, InductiveInvariant

THEOREM Correctness == Spec => []ResultCorrect
```

7.3 Divisibility Definition

In Shoup [Sho08] we find the divisibility definition.

Given the integers, a and b

$$a \text{ divides } b \implies az = b \text{ for some } z$$

so or all a,b, and c

$$a|a, \quad 1|a, \quad and \quad a|0$$

because $a \cdot 1 = a$, $1 \cdot a = a$, and $a \cdot 0 = 0$

$$0|a \iff a = 0$$

$$a|b \iff -a|b \iff a|-b$$

$$a|b \text{ and } a|c \implies a|(b+c)$$

$$a|b \text{ and } b|c \implies a|c$$

$$a|b \text{ and } b \neq 0 \implies 1 \leq |a| \leq |b|$$

$$az = b \neq 0 \text{ and } a \neq 0 \text{ and } z \neq 0 \implies |a| \geq 1 \text{ and } |z| \geq 1$$

$$a|b \text{ and } b|a \implies a = \pm b$$

proof:

$$a|b \implies |a| \leq |b|; b|a \implies |b| \leq |a|; \text{ therefore } |a| = |b| \implies a = \pm b$$

$$a|1 \iff a = \pm 1$$

Chapter 8

COQ proof of GCD

8.1 Basics of the Calculus of Constructions

Coquand [Coqu86] [Wiki17] defines the Calculus of Constructions which can be considered an extension of the Curry-Howard Isomorphism. The components are

8.1.1 Terms

A *term* in the calculus of constructions is constructed using the following rules:

- **T** is a term (also called *Type*)
- **P** is a term (also called *Prop*, the type of all propositions)
- Variables $(x, y, \ldots)$ are terms
- if **A** and **B** are terms, then so are
  - (A, B)
  - $(\lambda x : A, B)$
  - $(\forall x : A, B)$

The calculus of constructions has five kinds of objects:

1. *proofs*, which are terms whose types are *propositions*
2. *propositions*, which are also known as *small types*
3. *predicates*, which are functions that return propositions
4. *large types*, which are the types of predicates. **P** is an example of a large type)
5. **T** itself, which is the type of large types.

8.1.2 Judgements

The calculus of constructions allows proving **typing judgements**

$$x_1 : A_1, x_2 : A_2, \ldots \vdash t : B$$

which can be read as the implication

$$\text{if variables } x_1, x_2, \ldots, \text{ have types } A_1, A_2, \ldots, \text{ then term } t \text{ has type } B$$

The valid judgements for the calculus of constructions are derivable from a set of inference rules. In the following, we use Γ to mean a sequence of type assignments $x_1 : A_1$, $x_2 : A_2, \ldots$, and we use **K** to mean either **P** or **T**. We shall write $A : B : C$ to mean "A has type B, and B has type C". We shall write $B(x := N)$ to mean the result of substituting the term N for the variable x in the term B.

An inference rule is written in the form

$$\frac{\mathbf{\Gamma} \vdash \mathbf{A} : \mathbf{B}}{\mathbf{\Gamma}' \vdash \mathbf{C} : \mathbf{D}}$$

which means

$$\text{if } \mathbf{\Gamma} \vdash \mathbf{A} : \mathbf{B} \text{ is a valid judgement, then so is } \mathbf{\Gamma}' \vdash \mathbf{C} : \mathbf{D}$$

8.1.3 Inference Rules

In Frade [Frad08] we find:

(axiom)	$() \vdash s_1 : s_2$	if $(s_1, s_2) \in A$
(start)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$	if $x \notin \operatorname{dom}(\Gamma)$
(weakening)	$\dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma, x : B \vdash M : A}$	if $x \notin \operatorname{dom}(\Gamma)$
(product)	$\dfrac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\prod x : A.B) : s_3}$	if $(s_1, s_2, s_3) \in \mathbb{R}$
(application)	$\dfrac{\Gamma \vdash M : (\prod x : A.B) \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$	
(abstraction)	$\dfrac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash (\prod x : A.B) : s}{\Gamma \vdash \lambda x : A.M : (\prod x : A.B)}$	
(conversion)	$\dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}$	if $A =_\beta B$

8.1.4 Defining Logical Operators

$$\begin{array}{rcll}
A \Rightarrow B & \equiv & \forall x : A.B & (x \notin B) \\
A \wedge B & \equiv & \forall C : P.(A \Rightarrow B \Rightarrow C) \Rightarrow C & \\
A \vee B & \equiv & \forall C : P.(A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C & \\
\neg A & \equiv & \forall C : P.(A \Rightarrow C) & \\
\exists x : A.B & \equiv & \forall C : P.(\forall x : A.(B \Rightarrow C)) \Rightarrow C &
\end{array}$$

8.1.5 Defining Types

The basic data types used in computer science can be defined within the Calculus of Constructions:

Booleans

$$\forall A : P.A \Rightarrow A \Rightarrow A$$

Naturals

$$\forall A : P.(A \Rightarrow A) \Rightarrow (A \Rightarrow A)$$

Product $A \times B$

$$A \wedge B$$

Disjoint Union $A + B$

$$A \vee B$$

Note that Booleans and Naturals are defined in the same way as in Church encoding. However additional problems raise from propositional extensionality and proof irrelevance.

8.2 Why does COQ have Prop?

From a stackexchange post [Stac17] we find the question:

"Coq has a type `Prop` of proof irrelevant propositions which are discarded during extraction. What are the reasons for having this if we use Coq only for proofs? Prop is impredicative, however, Coq automatically infers universe indexes and we can use `Type(i)` instead everywhere. It seems `Prop` complicates everything a lot."

Prop is very useful for program extraction because it allows us to delete parts of code that are useless. For example, to extract a sorting algorithm we would prove the statement "for every list l there is a list k such that k is ordered and k is a permutation of l". If we write this down in Coq and extract without using Prop, we will get:

1. "for all l there is a k" which gives us a map **sort** which takes lists to lists,
2. "such that k is ordered" will give a function **verify** which runs through k and checks that it is sorted, and
3. "k is a permutation of l will give a permutation **p1** which takes l to k. Note that **p1** is not just a mapping, but also the inverse mapping together with programs verifying that the two maps really are inverses.

While the extra stuff is not totally useless, in many applications we want to get ride of it and keep just **sort**. This can be accomplished if we use `Prop` to state "k is ordered" and "k is a permutation of l", but *not* "for all l there is k".

In general, a common way to extract code is to consider a statement of the form

$$\forall x : A.\exists y : B.\phi(x, y)$$

where x is input, y is output, and $\phi(x, y)$ explains what it means for y to be a correct output. (In the above example A and B are the types of lists and $\phi(l, k)$ is "k is ordered and k is a permutation of l.") if ϕ is in `Prop` then extraction gives a map $f : A \Rightarrow B$ such that $\phi(x, f(x))$ holds for all $x \in A$. If ϕ is in `Set` then we also get a function g such that $g(x)$ is the proof that $\phi(x, f(x))$ holds, for all $x \in A$. Often the proof is computationally useless and we prefer to get rid of it, especially when it is nested deeply inside some other statement. `Prop` gives use the possibility to do so.

There is a question whether we could avoid `Prop` altogether by automatically optimizing away "useless extracted code". To some extent we can do that, for instance all code extracted from the negative fragment of logic (stuff build from the empty type, unit type, products) is useless as it just shuffles around the unit. But there are genuine design decisions one has to make when using `Prop`. Here is a simple example, where $\sum$ means that we are in `Type` and $\exists$ means we are in `Prop`. If we extract from

$$\prod_{n:N} \sum_{b:[0,1]} \sum_{k:N} \quad n = 2 \cdot k + b$$

we will get an inductivea program which decomposes n into its lowest bit b and the remaining bits k, i.e., it computes everything. If we extract from

$$\prod_{n:N} \sum_{b:[0,1]} \exists_{k:N} \quad n = 2 \cdot k + b$$

then the program will only compute the lowest bit b. The machine cannot tell which is the correct one, the user has to tell it what he wants.

8.3 Source code of COQ GCD Proof

This is the proof of GCD [Coqu16a] in the COQ [Coqu16] sources:

```
Library Coq.ZArith.Znumtheory

Require Import ZArith_base.
Require Import ZArithRing.
Require Import Zcomplements.
Require Import Zdiv.
Require Import Wf_nat.

For compatibility reasons, this Open Scope isn't local as it should

Open Scope Z_scope.

This file contains some notions of number theory upon Z numbers:

    a divisibility predicate Z.divide
    a gcd predicate gcd
    Euclid algorithm euclid
    a relatively prime predicate rel_prime
    a prime predicate prime
    properties of the efficient Z.gcd function
```

```
Notation Zgcd := Z.gcd (compat "8.3").
Notation Zggcd := Z.ggcd (compat "8.3").
Notation Zggcd_gcd := Z.ggcd_gcd (compat "8.3").
Notation Zggcd_correct_divisors := Z.ggcd_correct_divisors (compat "8.3").
Notation Zgcd_divide_l := Z.gcd_divide_l (compat "8.3").
Notation Zgcd_divide_r := Z.gcd_divide_r (compat "8.3").
Notation Zgcd_greatest := Z.gcd_greatest (compat "8.3").
Notation Zgcd_nonneg := Z.gcd_nonneg (compat "8.3").
Notation Zggcd_opp := Z.ggcd_opp (compat "8.3").

The former specialized inductive predicate Z.divide is now a generic existential predicate.

Notation Zdivide := Z.divide (compat "8.3").

Its former constructor is now a pseudo-constructor.

Definition Zdivide_intro a b q (H:b=q*a) : Z.divide a b := ex_intro _ q H.

Results concerning divisibility

Notation Zdivide_refl := Z.divide_refl (compat "8.3").
Notation Zone_divide := Z.divide_1_l (compat "8.3").
Notation Zdivide_0 := Z.divide_0_r (compat "8.3").
Notation Zmult_divide_compat_l := Z.mul_divide_mono_l (compat "8.3").
Notation Zmult_divide_compat_r := Z.mul_divide_mono_r (compat "8.3").
Notation Zdivide_plus_r := Z.divide_add_r (compat "8.3").
Notation Zdivide_minus_l := Z.divide_sub_r (compat "8.3").
Notation Zdivide_mult_l := Z.divide_mul_l (compat "8.3").
Notation Zdivide_mult_r := Z.divide_mul_r (compat "8.3").
Notation Zdivide_factor_r := Z.divide_factor_l (compat "8.3").
Notation Zdivide_factor_l := Z.divide_factor_r (compat "8.3").

Lemma Zdivide_opp_r a b : (a | b) -> (a | - b).

Lemma Zdivide_opp_r_rev a b : (a | - b) -> (a | b).

Lemma Zdivide_opp_l a b : (a | b) -> (- a | b).

Lemma Zdivide_opp_l_rev a b : (- a | b) -> (a | b).

Theorem Zdivide_Zabs_l a b : (Z.abs a | b) -> (a | b).

Theorem Zdivide_Zabs_inv_l a b : (a | b) -> (Z.abs a | b).

Hint Resolve Z.divide_refl Z.divide_1_l Z.divide_0_r: zarith.
Hint Resolve Z.mul_divide_mono_l Z.mul_divide_mono_r: zarith.
Hint Resolve Z.divide_add_r Zdivide_opp_r Zdivide_opp_r_rev Zdivide_opp_l
  Zdivide_opp_l_rev Z.divide_sub_r Z.divide_mul_l Z.divide_mul_r
  Z.divide_factor_l Z.divide_factor_r: zarith.

Auxiliary result.

Lemma Zmult_one x y : x >= 0 -> x * y = 1 -> x = 1.
```

Only 1 and -1 divide 1.

```
Notation Zdivide_1 := Z.divide_1_r (compat "8.3").
```

If a divides b and b divides a then a is b or -b.

```
Notation Zdivide_antisym := Z.divide_antisym (compat "8.3").
Notation Zdivide_trans := Z.divide_trans (compat "8.3").
```

If a divides b and b<>0 then |a| <= |b|.

```
Lemma Zdivide_bounds a b : (a | b) -> b <> 0 -> Z.abs a <= Z.abs b.
```

Z.divide can be expressed using Z.modulo.

```
Lemma Zmod_divide : forall a b, b<>0 -> a mod b = 0 -> (b | a).

Lemma Zdivide_mod : forall a b, (b | a) -> a mod b = 0.
```

Z.divide is hence decidable

```
Lemma Zdivide_dec a b : {(a | b)} + {~ (a | b)}.

Theorem Zdivide_Zdiv_eq a b : 0 < a -> (a | b) -> b = a * (b / a).

Theorem Zdivide_Zdiv_eq_2 a b c :
 0 < a -> (a | b) -> (c * b) / a = c * (b / a).

Theorem Zdivide_le: forall a b : Z,
 0 <= a -> 0 < b -> (a | b) -> a <= b.

Theorem Zdivide_Zdiv_lt_pos a b :
 1 < a -> 0 < b -> (a | b) -> 0 < b / a < b .

Lemma Zmod_div_mod n m a:
 0 < n -> 0 < m -> (n | m) -> a mod n = (a mod m) mod n.

Lemma Zmod_divide_minus a b c:
 0 < b -> a mod b = c -> (b | a - c).

Lemma Zdivide_mod_minus a b c:
 0 <= c < b -> (b | a - c) -> a mod b = c.
```

Greatest common divisor (gcd).
There is no unicity of the gcd; hence we define the predicate Zis_gcd a b g expressing that g is a gcd of a and

```
Inductive Zis_gcd (a b g:Z) : Prop :=
 Zis_gcd_intro :
  (g | a) ->
  (g | b) ->
  (forall x, (x | a) -> (x | b) -> (x | g)) ->
  Zis_gcd a b g.
```

Trivial properties of gcd

```
Lemma Zis_gcd_sym : forall a b d, Zis_gcd a b d -> Zis_gcd b a d.

Lemma Zis_gcd_0 : forall a, Zis_gcd a 0 a.

Lemma Zis_gcd_1 : forall a, Zis_gcd a 1 1.

Lemma Zis_gcd_refl : forall a, Zis_gcd a a a.

Lemma Zis_gcd_minus : forall a b d, Zis_gcd a (- b) d -> Zis_gcd b a d.

Lemma Zis_gcd_opp : forall a b d, Zis_gcd a b d -> Zis_gcd b a (- d).

Lemma Zis_gcd_0_abs a : Zis_gcd 0 a (Z.abs a).

Hint Resolve Zis_gcd_sym Zis_gcd_0 Zis_gcd_minus Zis_gcd_opp: zarith.

Theorem Zis_gcd_unique: forall a b c d : Z,
 Zis_gcd a b c -> Zis_gcd a b d -> c = d \/ c = (- d).
```

Extended Euclid algorithm.
Euclid's algorithm to compute the gcd mainly relies on the following property.

```
Lemma Zis_gcd_for_euclid :
  forall a b d q:Z, Zis_gcd b (a - q * b) d -> Zis_gcd a b d.

Lemma Zis_gcd_for_euclid2 :
  forall b d q r:Z, Zis_gcd r b d -> Zis_gcd b (b * q + r) d.
```

We implement the extended version of Euclid's algorithm, i.e. the one computing Bezout's coefficients as it comp

```
Section extended_euclid_algorithm.

  Variables a b : Z.
```

The specification of Euclid's algorithm is the existence of u, v and d such that ua+vb=d and (gcd a b d).

```
  Inductive Euclid : Set :=
    Euclid_intro :
    forall u v d:Z, u * a + v * b = d -> Zis_gcd a b d -> Euclid.
```

The recursive part of Euclid's algorithm uses well-founded recursion of non-negative integers. It maintains 6 in

```
  Lemma euclid_rec :
    forall v3:Z,
      0 <= v3 ->
      forall u1 u2 u3 v1 v2:Z,
        u1 * a + u2 * b = u3 ->
        v1 * a + v2 * b = v3 ->
        (forall d:Z, Zis_gcd u3 v3 d -> Zis_gcd a b d) -> Euclid.
```

We get Euclid's algorithm by applying euclid_rec on 1,0,a,0,1,b when b>=0 and 1,0,a,0,-1,-b when b<0.

```
  Lemma euclid : Euclid.

End extended_euclid_algorithm.

Theorem Zis_gcd_uniqueness_apart_sign :
  forall a b d d':Z, Zis_gcd a b d -> Zis_gcd a b d' -> d = d' \/ d = - d'.
```

Bezout's coefficients

```
Inductive Bezout (a b d:Z) : Prop :=
  Bezout_intro : forall u v:Z, u * a + v * b = d -> Bezout a b d.
```

Existence of Bezout's coefficients for the gcd of a and b

```
Lemma Zis_gcd_bezout : forall a b d:Z, Zis_gcd a b d -> Bezout a b d.
```

gcd of ca and cb is c gcd(a,b).

```
Lemma Zis_gcd_mult :
  forall a b c d:Z, Zis_gcd a b d -> Zis_gcd (c * a) (c * b) (c * d).
```

Relative primality

```
Definition rel_prime (a b:Z) : Prop := Zis_gcd a b 1.
```

Bezout's theorem: a and b are relatively prime if and only if there exist u and v such that ua+vb = 1.

```
Lemma rel_prime_bezout : forall a b:Z, rel_prime a b -> Bezout a b 1.

Lemma bezout_rel_prime : forall a b:Z, Bezout a b 1 -> rel_prime a b.
```

Gauss's theorem: if a divides bc and if a and b are relatively prime, then a divides c.

```
Theorem Gauss : forall a b c:Z, (a | b * c) -> rel_prime a b -> (a | c).
```

If a is relatively prime to b and c, then it is to bc

```
Lemma rel_prime_mult :
  forall a b c:Z, rel_prime a b -> rel_prime a c -> rel_prime a (b * c).

Lemma rel_prime_cross_prod :
  forall a b c d:Z,
    rel_prime a b ->
    rel_prime c d -> b > 0 -> d > 0 -> a * d = b * c -> a = c /\ b = d.
```

After factorization by a gcd, the original numbers are relatively prime.

```
Lemma Zis_gcd_rel_prime :
  forall a b g:Z,
    b > 0 -> g >= 0 -> Zis_gcd a b g -> rel_prime (a / g) (b / g).

Theorem rel_prime_sym: forall a b, rel_prime a b -> rel_prime b a.

Theorem rel_prime_div: forall p q r,
```

```
 rel_prime p q -> (r | p) -> rel_prime r q.

Theorem rel_prime_1: forall n, rel_prime 1 n.

Theorem not_rel_prime_0: forall n, 1 < n -> ~ rel_prime 0 n.

Theorem rel_prime_mod: forall p q, 0 < q ->
 rel_prime p q -> rel_prime (p mod q) q.

Theorem rel_prime_mod_rev: forall p q, 0 < q ->
 rel_prime (p mod q) q -> rel_prime p q.

Theorem Zrel_prime_neq_mod_0: forall a b, 1 < b -> rel_prime a b -> a mod b <> 0.

Primality

Inductive prime (p:Z) : Prop :=
  prime_intro :
    1 < p -> (forall n:Z, 1 <= n < p -> rel_prime n p) -> prime p.

The sole divisors of a prime number p are -1, 1, p and -p.

Lemma prime_divisors :
  forall p:Z,
    prime p -> forall a:Z, (a | p) -> a = -1 \/ a = 1 \/ a = p \/ a = - p.

A prime number is relatively prime with any number it does not divide

Lemma prime_rel_prime :
  forall p:Z, prime p -> forall a:Z, ~ (p | a) -> rel_prime p a.

Hint Resolve prime_rel_prime: zarith.

As a consequence, a prime number is relatively prime with smaller numbers

Theorem rel_prime_le_prime:
 forall a p, prime p -> 1 <= a < p -> rel_prime a p.

If a prime p divides ab then it divides either a or b

Lemma prime_mult :
  forall p:Z, prime p -> forall a b:Z, (p | a * b) -> (p | a) \/ (p | b).

Lemma not_prime_0: ~ prime 0.

Lemma not_prime_1: ~ prime 1.

Lemma prime_2: prime 2.

Theorem prime_3: prime 3.

Theorem prime_ge_2 p : prime p -> 2 <= p.

Definition prime' p := 1<p /\ (forall n, 1<n<p -> ~ (n|p)).
```

```
Lemma Z_0_1_more x : 0<=x -> x=0 \/ x=1 \/ 1<x.

Theorem prime_alt p : prime' p <-> prime p.

Theorem square_not_prime: forall a, ~ prime (a * a).

Theorem prime_div_prime: forall p q,
 prime p -> prime q -> (p | q) -> p = q.
```

we now prove that Z.gcd is indeed a gcd in the sense of Zis_gcd.

```
Notation Zgcd_is_pos := Z.gcd_nonneg (compat "8.3").

Lemma Zgcd_is_gcd : forall a b, Zis_gcd a b (Z.gcd a b).

Theorem Zgcd_spec : forall x y : Z, {z : Z | Zis_gcd x y z /\ 0 <= z}.

Theorem Zdivide_Zgcd: forall p q r : Z,
 (p | q) -> (p | r) -> (p | Z.gcd q r).

Theorem Zis_gcd_gcd: forall a b c : Z,
 0 <= c -> Zis_gcd a b c -> Z.gcd a b = c.

Notation Zgcd_inv_0_l := Z.gcd_eq_0_l (compat "8.3").
Notation Zgcd_inv_0_r := Z.gcd_eq_0_r (compat "8.3").

Theorem Zgcd_div_swap0 : forall a b : Z,
 0 < Z.gcd a b ->
 0 < b ->
 (a / Z.gcd a b) * b = a * (b/Z.gcd a b).

Theorem Zgcd_div_swap : forall a b c : Z,
 0 < Z.gcd a b ->
 0 < b ->
 (c * a) / Z.gcd a b * b = c * a * (b/Z.gcd a b).

Notation Zgcd_comm := Z.gcd_comm (compat "8.3").

Lemma Zgcd_ass a b c : Z.gcd (Z.gcd a b) c = Z.gcd a (Z.gcd b c).

Notation Zgcd_Zabs := Z.gcd_abs_l (compat "8.3").
Notation Zgcd_0 := Z.gcd_0_r (compat "8.3").
Notation Zgcd_1 := Z.gcd_1_r (compat "8.3").

Hint Resolve Z.gcd_0_r Z.gcd_1_r : zarith.

Theorem Zgcd_1_rel_prime : forall a b,
 Z.gcd a b = 1 <-> rel_prime a b.

Definition rel_prime_dec: forall a b,
 { rel_prime a b }+{ ~ rel_prime a b }.

Definition prime_dec_aux:
```

```
 forall p m,
  { forall n, 1 < n < m -> rel_prime n p } +
  { exists n, 1 < n < m /\ ~ rel_prime n p }.

Definition prime_dec: forall p, { prime p }+{ ~ prime p }.

Theorem not_prime_divide:
 forall p, 1 < p -> ~ prime p -> exists n, 1 < n < p /\ (n | p).
```

Chapter 9

LEAN proof of GCD

This is the proof of GCD [Avig14] in the LEAN [Avig16] sources:

```
/-
Copyright (c) 2014 Jeremy Avigad. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Jeremy Avigad, Leonardo de Moura

Definitions and properties of gcd, lcm, and coprime.
-/
import .div
open eq.ops well_founded decidable prod

namespace nat

/- gcd -/

private definition pair_nat.lt : nat  nat  nat  nat  Prop := measure pr
private definition pair_nat.lt.wf : well_founded pair_nat.lt :=
intro_k (measure.wf pr) 20  -- we use intro_k to be able to execute gcd efficiently in the kernel

local attribute pair_nat.lt.wf [instance]      -- instance will not be saved in .olean
local infixl `  `:50 := pair_nat.lt

private definition gcd.lt.dec (x y : nat) : (succ y, x % succ y)  (x, succ y) :=
!mod_lt (succ_pos y)

definition gcd.F :  (p : nat  nat), ( p : nat  nat, p  p  nat)  nat
| (x, 0)      f := x
| (x, succ y) f := f (succ y, x % succ y) !gcd.lt.dec

definition gcd (x y : nat) := fix gcd.F (x, y)

theorem gcd_zero_right [simp] (x : nat) : gcd x 0 = x := rfl

theorem gcd_succ [simp] (x y : nat) : gcd x (succ y) = gcd (succ y) (x % succ y) :=
well_founded.fix_eq gcd.F (x, succ y)
```

```
theorem gcd_one_right (n : ) : gcd n 1 = 1 :=
calc gcd n 1 = gcd 1 (n % 1)  : gcd_succ
         ... = gcd 1 0        : mod_one

theorem gcd_def (x : ) :  (y : ), gcd x y = if y = 0 then x else gcd y (x % y)
| 0        := !gcd_zero_right
| (succ y) := !gcd_succ  (if_neg !succ_ne_zero)


theorem gcd_self :  (n : ), gcd n n = n
| 0        := rfl
| (succ n) := calc
    gcd (succ n) (succ n) = gcd (succ n) (succ n % succ n) : gcd_succ
                    ...   = gcd (succ n) 0                  : mod_self

theorem gcd_zero_left :  (n : ), gcd 0 n = n
| 0        := rfl
| (succ n) := calc
    gcd 0 (succ n) = gcd (succ n) (0 % succ n) : gcd_succ
              ... = gcd (succ n) 0             : zero_mod

theorem gcd_of_pos (m : ) {n : } (H : n > 0) : gcd m n = gcd n (m % n) :=
gcd_def m n  if_neg (ne_zero_of_pos H)

theorem gcd_rec (m n : ) : gcd m n = gcd n (m % n) :=
by_cases_zero_pos n
  (calc
         m = gcd 0 m       : gcd_zero_left
        ... = gcd 0 (m % 0) : mod_zero)
  (take n, assume H : 0 < n, gcd_of_pos m H)

theorem gcd.induction {P :     Prop}
                   (m n : )
                   (H0 : m, P m 0)
                   (H1 : m n, 0 < n  P n (m % n)  P m n) :
                 P m n :=
induction (m, n) (prod.rec (m, nat.rec ( IH, H0 m)
   ( n v (IH : p, p  (m, succ n)  P (pr p) (pr p)),
      H1 m (succ n) !succ_pos (IH _ !gcd.lt.dec))))

theorem gcd_dvd (m n : ) : (gcd m n  m)  (gcd m n  n) :=
gcd.induction m n
  (take m, and.intro (!one_mul  !dvd_mul_left) !dvd_zero)
  (take m n (npos : 0 < n), and.rec
     (assume (IH : gcd n (m % n)  n) (IH : gcd n (m % n)  (m % n)),
    have H : (gcd n (m % n)  (m / n * n + m % n)), from
      dvd_add (dvd.trans IH !dvd_mul_left) IH,
    have H1 : (gcd n (m % n)  m), from !eq_div_mul_add_mod  H,
    show (gcd m n  m)  (gcd m n  n), from !gcd_rec  (and.intro H1 IH)))

theorem gcd_dvd_left (m n : ) : gcd m n  m := and.left !gcd_dvd

theorem gcd_dvd_right (m n : ) : gcd m n  n := and.right !gcd_dvd
```

```
theorem dvd_gcd {m n k : } : k  m  k  n  k  gcd m n :=
gcd.induction m n (take m, imp.intro)
  (take m n (npos : n > 0)
    (IH : k  n  k  m % n  k  gcd n (m % n))
    (H1 : k  m) (H2 : k  n),
    have H3 : k  m / n * n + m % n, from !eq_div_mul_add_mod  H1,
    have H4 : k  m % n, from nat.dvd_of_dvd_add_left H3 (dvd.trans H2 !dvd_mul_left),
    !gcd_rec  IH H2 H4)

theorem gcd.comm (m n : ) : gcd m n = gcd n m :=
dvd.antisymm
  (dvd_gcd !gcd_dvd_right !gcd_dvd_left)
  (dvd_gcd !gcd_dvd_right !gcd_dvd_left)

theorem gcd.assoc (m n k : ) : gcd (gcd m n) k = gcd m (gcd n k) :=
dvd.antisymm
  (dvd_gcd
    (dvd.trans !gcd_dvd_left !gcd_dvd_left)
    (dvd_gcd (dvd.trans !gcd_dvd_left !gcd_dvd_right) !gcd_dvd_right))
  (dvd_gcd
    (dvd_gcd !gcd_dvd_left (dvd.trans !gcd_dvd_right !gcd_dvd_left))
    (dvd.trans !gcd_dvd_right !gcd_dvd_right))

theorem gcd_one_left (m : ) : gcd 1 m = 1 :=
!gcd.comm  !gcd_one_right

theorem gcd_mul_left (m n k : ) : gcd (m * n) (m * k) = m * gcd n k :=
gcd.induction n k
  (take n, calc gcd (m * n) (m * 0) = gcd (m * n) 0 : mul_zero)
  (take n k,
    assume H : 0 < k,
    assume IH : gcd (m * k) (m * (n % k)) = m * gcd k (n % k),
    calc
      gcd (m * n) (m * k) = gcd (m * k) (m * n % (m * k)) : !gcd_rec
                      ... = gcd (m * k) (m * (n % k))     : mul_mod_mul_left
                      ... = m * gcd k (n % k)             : IH
                      ... = m * gcd n k                   : !gcd_rec)

theorem gcd_mul_right (m n k : ) : gcd (m * n) (k * n) = gcd m k * n :=
calc
  gcd (m * n) (k * n) = gcd (n * m) (k * n) : mul.comm
                  ... = gcd (n * m) (n * k) : mul.comm
                  ... = n * gcd m k         : gcd_mul_left
                  ... = gcd m k * n         : mul.comm

theorem gcd_pos_of_pos_left {m : } (n : ) (mpos : m > 0) : gcd m n > 0 :=
pos_of_dvd_of_pos !gcd_dvd_left mpos

theorem gcd_pos_of_pos_right (m : ) {n : } (npos : n > 0) : gcd m n > 0 :=
pos_of_dvd_of_pos !gcd_dvd_right npos

theorem eq_zero_of_gcd_eq_zero_left {m n : } (H : gcd m n = 0) : m = 0 :=
or.elim (eq_zero_or_pos m)
  (assume H1, H1)
```

```
  (assume H1 : m > 0, absurd H (ne_of_lt (!gcd_pos_of_pos_left H1)))

theorem eq_zero_of_gcd_eq_zero_right {m n : } (H : gcd m n = 0) : n = 0 :=
eq_zero_of_gcd_eq_zero_left (!gcd.comm  H)

theorem gcd_div {m n k : } (H1 : k  m) (H2 : k  n) :
  gcd (m / k) (n / k) = gcd m n / k :=
or.elim (eq_zero_or_pos k)
  (assume H3 : k = 0, by subst k; rewrite *nat.div_zero)
  (assume H3 : k > 0, (nat.div_eq_of_eq_mul_left H3 (calc
        gcd m n = gcd m (n / k * k)             : nat.div_mul_cancel H2
            ... = gcd (m / k * k) (n / k * k) : nat.div_mul_cancel H1
            ... = gcd (m / k) (n / k) * k     : gcd_mul_right)))

theorem gcd_dvd_gcd_mul_left (m n k : ) : gcd m n  gcd (k * m) n :=
dvd_gcd (dvd.trans !gcd_dvd_left !dvd_mul_left) !gcd_dvd_right

theorem gcd_dvd_gcd_mul_right (m n k : ) : gcd m n  gcd (m * k) n :=
!mul.comm  !gcd_dvd_gcd_mul_left

theorem gcd_dvd_gcd_mul_left_right (m n k : ) : gcd m n  gcd m (k * n) :=
dvd_gcd  !gcd_dvd_left (dvd.trans !gcd_dvd_right !dvd_mul_left)

theorem gcd_dvd_gcd_mul_right_right (m n k : ) : gcd m n  gcd m (n * k) :=
!mul.comm  !gcd_dvd_gcd_mul_left_right

/- lcm -/

definition lcm (m n : ) :  := m * n / (gcd m n)

theorem lcm.comm (m n : ) : lcm m n = lcm n m :=
calc
  lcm m n = m * n / gcd m n : rfl
      ... = n * m / gcd m n : mul.comm
      ... = n * m / gcd n m : gcd.comm
      ... = lcm n m         : rfl

theorem lcm_zero_left (m : ) : lcm 0 m = 0 :=
calc
  lcm 0 m = 0 * m / gcd 0 m : rfl
      ... = 0 / gcd 0 m     : zero_mul
      ... = 0               : nat.zero_div

theorem lcm_zero_right (m : ) : lcm m 0 = 0 := !lcm.comm  !lcm_zero_left

theorem lcm_one_left (m : ) : lcm 1 m = m :=
calc
  lcm 1 m = 1 * m / gcd 1 m : rfl
      ... = m / gcd 1 m     : one_mul
      ... = m / 1           : gcd_one_left
      ... = m               : nat.div_one

theorem lcm_one_right (m : ) : lcm m 1 = m := !lcm.comm  !lcm_one_left
```

```
theorem lcm_self (m : ) : lcm m m = m :=
have H : m * m / m = m, from
  by_cases_zero_pos m !nat.div_zero (take m, assume H1 : m > 0, !nat.mul_div_cancel H1),
calc
  lcm m m = m * m / gcd m m : rfl
      ... = m * m / m       : gcd_self
      ... = m                : H

theorem dvd_lcm_left (m n : ) : m  lcm m n :=
have H : lcm m n = m * (n / gcd m n), from nat.mul_div_assoc _ !gcd_dvd_right,
dvd.intro H

theorem dvd_lcm_right (m n : ) : n  lcm m n :=
!lcm.comm  !dvd_lcm_left

theorem gcd_mul_lcm (m n : ) : gcd m n * lcm m n = m * n :=
eq.symm (nat.eq_mul_of_div_eq_right (dvd.trans !gcd_dvd_left !dvd_mul_right) rfl)

theorem lcm_dvd {m n k : } (H1 : m  k) (H2 : n  k) : lcm m n  k :=
or.elim (eq_zero_or_pos k)
  (assume kzero : k = 0, !kzero  !dvd_zero)
  (assume kpos : k > 0,
    have mpos : m > 0, from pos_of_dvd_of_pos H1 kpos,
    have npos : n > 0, from pos_of_dvd_of_pos H2 kpos,
    have gcd_pos : gcd m n > 0, from !gcd_pos_of_pos_left mpos,
    obtain p (km : k = m * p), from exists_eq_mul_right_of_dvd H1,
    obtain q (kn : k = n * q), from exists_eq_mul_right_of_dvd H2,
    have ppos : p > 0, from pos_of_mul_pos_left (km  kpos),
    have qpos : q > 0, from pos_of_mul_pos_left (kn  kpos),
    have H3 : p * q * (m * n * gcd p q) = p * q * (gcd m n * k), from
    calc
      p * q * (m * n * gcd p q)
            = m * p * (n * q * gcd p q)       : by rewrite [*mul.assoc, *mul.left_comm q,
                                                             mul.left_comm p]
        ... = k * (k * gcd p q)               : by rewrite [-kn, -km]
        ... = k * gcd (k * p) (k * q)         : by rewrite gcd_mul_left
        ... = k * gcd (n * q * p) (m * p * q) : by rewrite [-kn, -km]
        ... = k * (gcd n m * (p * q))         : by rewrite [*mul.assoc, mul.comm q, gcd_mul_right]
        ... = p * q * (gcd m n * k)           : by rewrite [mul.comm, mul.comm (gcd n m), gcd.comm,
                                                             *mul.assoc],
    have H4 : m * n * gcd p q = gcd m n * k,
      from !eq_of_mul_eq_mul_left (mul_pos ppos qpos) H3,
    have H5 : gcd m n * (lcm m n * gcd p q) = gcd m n * k,
      from !mul.assoc  !gcd_mul_lcm  H4,
    have H6 : lcm m n * gcd p q = k,
      from !eq_of_mul_eq_mul_left gcd_pos H5,
    dvd.intro H6)

theorem lcm.assoc (m n k : ) : lcm (lcm m n) k = lcm m (lcm n k) :=
dvd.antisymm
  (lcm_dvd
    (lcm_dvd !dvd_lcm_left (dvd.trans !dvd_lcm_left !dvd_lcm_right))
    (dvd.trans !dvd_lcm_right !dvd_lcm_right))
  (lcm_dvd
```

```
    (dvd.trans !dvd_lcm_left !dvd_lcm_left)
    (lcm_dvd (dvd.trans !dvd_lcm_right !dvd_lcm_left) !dvd_lcm_right))

/- coprime -/

definition coprime [reducible] (m n : ) : Prop := gcd m n = 1

lemma gcd_eq_one_of_coprime {m n : } : coprime m n  gcd m n = 1 :=
 h, h

theorem coprime_swap {m n : } (H : coprime n m) : coprime m n :=
!gcd.comm  H

theorem dvd_of_coprime_of_dvd_mul_right {m n k : } (H1 : coprime k n) (H2 : k  m * n) : k  m :=
have H3 : gcd (m * k) (m * n) = m, from
  calc
    gcd (m * k) (m * n) = m * gcd k n : gcd_mul_left
                    ... = m * 1       : H1
                    ... = m           : mul_one,
have H4 : (k  gcd (m * k) (m * n)), from dvd_gcd !dvd_mul_left H2,
H3  H4

theorem dvd_of_coprime_of_dvd_mul_left {m n k : } (H1 : coprime k m) (H2 : k  m * n) : k  n :=
dvd_of_coprime_of_dvd_mul_right H1 (!mul.comm  H2)

theorem gcd_mul_left_cancel_of_coprime {k : } (m : ) {n : } (H : coprime k n) :
   gcd (k * m) n = gcd m n :=
have H1 : coprime (gcd (k * m) n) k, from
  calc
    gcd (gcd (k * m) n) k
         = gcd (k * gcd 1 m) n : by rewrite [-gcd_mul_left, mul_one, gcd.comm, gcd.assoc]
     ... = 1                   : by rewrite [gcd_one_left, mul_one, coprime at H, H],
dvd.antisymm
  (dvd_gcd (dvd_of_coprime_of_dvd_mul_left H1 !gcd_dvd_left) !gcd_dvd_right)
  (dvd_gcd (dvd.trans !gcd_dvd_left !dvd_mul_left) !gcd_dvd_right)

theorem gcd_mul_right_cancel_of_coprime (m : ) {k n : } (H : coprime k n) :
   gcd (m * k) n = gcd m n :=
!mul.comm  !gcd_mul_left_cancel_of_coprime H

theorem gcd_mul_left_cancel_of_coprime_right {k m : } (n : ) (H : coprime k m) :
   gcd m (k * n) = gcd m n :=
!gcd.comm  !gcd.comm  !gcd_mul_left_cancel_of_coprime H

theorem gcd_mul_right_cancel_of_coprime_right {k m : } (n : ) (H : coprime k m) :
   gcd m (n * k) = gcd m n :=
!gcd.comm  !gcd.comm  !gcd_mul_right_cancel_of_coprime H

theorem coprime_div_gcd_div_gcd {m n : } (H : gcd m n > 0) :
  coprime (m / gcd m n) (n / gcd m n) :=
calc
  gcd (m / gcd m n) (n / gcd m n) = gcd m n / gcd m n : gcd_div !gcd_dvd_left !gcd_dvd_right
     ... = 1 : nat.div_self H
```

```
theorem not_coprime_of_dvd_of_dvd {m n d : } (dgt1 : d > 1) (Hm : d  m) (Hn : d  n) :
   coprime m n :=
assume co : coprime m n,
have d  gcd m n, from dvd_gcd Hm Hn,
have d  1, by rewrite [coprime at co, co at this]; apply this,
have d  1, from le_of_dvd dec_trivial this,
show false, from not_lt_of_ge `d  1` `d > 1`

theorem exists_coprime {m n : } (H : gcd m n > 0) :
  exists m' n', coprime m' n'  m = m' * gcd m n  n = n' * gcd m n :=
have H1 : m = (m / gcd m n) * gcd m n, from (nat.div_mul_cancel !gcd_dvd_left),
have H2 : n = (n / gcd m n) * gcd m n, from (nat.div_mul_cancel !gcd_dvd_right),
exists.intro _ (exists.intro _ (and.intro (coprime_div_gcd_div_gcd H) (and.intro H1 H2)))

theorem coprime_mul {m n k : } (H1 : coprime m k) (H2 : coprime n k) : coprime (m * n) k :=
calc
  gcd (m * n) k = gcd n k : !gcd_mul_left_cancel_of_coprime H1
            ... = 1       : H2

theorem coprime_mul_right {k m n : } (H1 : coprime k m) (H2 : coprime k n) : coprime k (m * n) :=
coprime_swap (coprime_mul (coprime_swap H1) (coprime_swap H2))

theorem coprime_of_coprime_mul_left {k m n : } (H : coprime (k * m) n) : coprime m n :=
have H1 : (gcd m n  gcd (k * m) n), from !gcd_dvd_gcd_mul_left,
eq_one_of_dvd_one (H  H1)

theorem coprime_of_coprime_mul_right {k m n : } (H : coprime (m * k) n) : coprime m n :=
coprime_of_coprime_mul_left (!mul.comm  H)

theorem coprime_of_coprime_mul_left_right {k m n : } (H : coprime m (k * n)) : coprime m n :=
coprime_swap (coprime_of_coprime_mul_left (coprime_swap H))

theorem coprime_of_coprime_mul_right_right {k m n : } (H : coprime m (n * k)) : coprime m n :=
coprime_of_coprime_mul_left_right (!mul.comm  H)

theorem comprime_one_left :  n, coprime 1 n :=
 n, !gcd_one_left

theorem comprime_one_right :  n, coprime n 1 :=
 n, !gcd_one_right

theorem exists_eq_prod_and_dvd_and_dvd {m n k : nat} (H : k  m * n) :
   m' n', k = m' * n'  m'  m  n'  n :=
or.elim (eq_zero_or_pos (gcd k m))
 (assume H1 : gcd k m = 0,
    have H2 : k = 0, from eq_zero_of_gcd_eq_zero_left H1,
    have H3 : m = 0, from eq_zero_of_gcd_eq_zero_right H1,
    have H4 : k = 0 * n, from H2  !zero_mul,
    have H5 : 0  m, from H3  !dvd.refl,
    have H6 : n  n, from !dvd.refl,
    exists.intro _ (exists.intro _ (and.intro H4 (and.intro H5 H6))))
  (assume H1 : gcd k m > 0,
    have H2 : gcd k m  k, from !gcd_dvd_left,
    have H3 : k / gcd k m  (m * n) / gcd k m, from nat.div_dvd_div H2 H,
```

```
    have H4 : (m * n) / gcd k m = (m / gcd k m) * n, from
      calc
        m * n / gcd k m = n * m / gcd k m   : mul.comm
                    ... = n * (m / gcd k m) : !nat.mul_div_assoc !gcd_dvd_right
                    ... = m / gcd k m * n   : mul.comm,
    have H5 : k / gcd k m  (m / gcd k m) * n, from H4  H3,
    have H6 : coprime (k / gcd k m) (m / gcd k m), from coprime_div_gcd_div_gcd H1,
    have H7 : k / gcd k m  n, from dvd_of_coprime_of_dvd_mul_left H6 H5,
    have H8 : k = gcd k m * (k / gcd k m), from (nat.mul_div_cancel' H2),
    exists.intro _ (exists.intro _ (and.intro H8 (and.intro !gcd_dvd_right H7))))

end nat
```

Chapter 10

Formal Pre- and Post-conditions

In Boldo [Bold11] we find an effort to verify floating point software using preconditions, postconditions, and assertions. Quoting:

“These conjectures can be described formally by annotations as follows.

```
/*@ requires \abs(x) <= 0x1p-5;
  @ ensures \abs(\result - \cos(x)) <= 0x1p-23;
  @*/
float my_cosine(float x) {
  //@ assert \abs(1.0 - x*x*0.5 - \cos(x)) < 0x1p-24;
  return 1.0f - x * x * 0.5f;
}
```

The *precondition*, introduced by **requires**, states that we expect argument x in the interval [-1/32; 1/32]. The *postcondition*, introduced by **ensures**, states that the distance between the value returned by the function, denoted by the keyword `\result`, and the model of the program, which is here the true mathematical cosine function denoted by `\cos` in ACSL, is not greater than 2^{-23}. It is important to notice that in annotations the operators like $+$ or $*$ denote operations on real numbers and not on floating-point numbers. In particular, there is no rounding error and no overflow in annotations, unlike in the early Leavens' proposal. The C variables of type `float`, like `x` and `\result` in this example, are interpreted as the real number they represent. Thus, the last annotation, given as an assertion inside the code, is a way to make explicit the reasoning we made above, making the total error the sum of the method error and the rounding error: it states that the method error is less than 2^{-24}. Again, it is thanks to the choice of having exact operations in the annotations that we are able to state a property of the method error.”

In Boldo [Bold07, Bold07a] we find 'search in an array' annotated:

```
/*@ requires \valid_range(t,0,n-1)
  @ ensures
  @  (0 <= \result < n => t[\result] == v) &&
  @  (\result == n =>
  @    \forall int i; 0 <= i < n => t[i] != v) */
int index(int t[], int n, int v) {
  int i = 0;
  /*@ invariant 0 <= i &&
    @  \forall int k; 0 <= k <i => t[k] != v
```

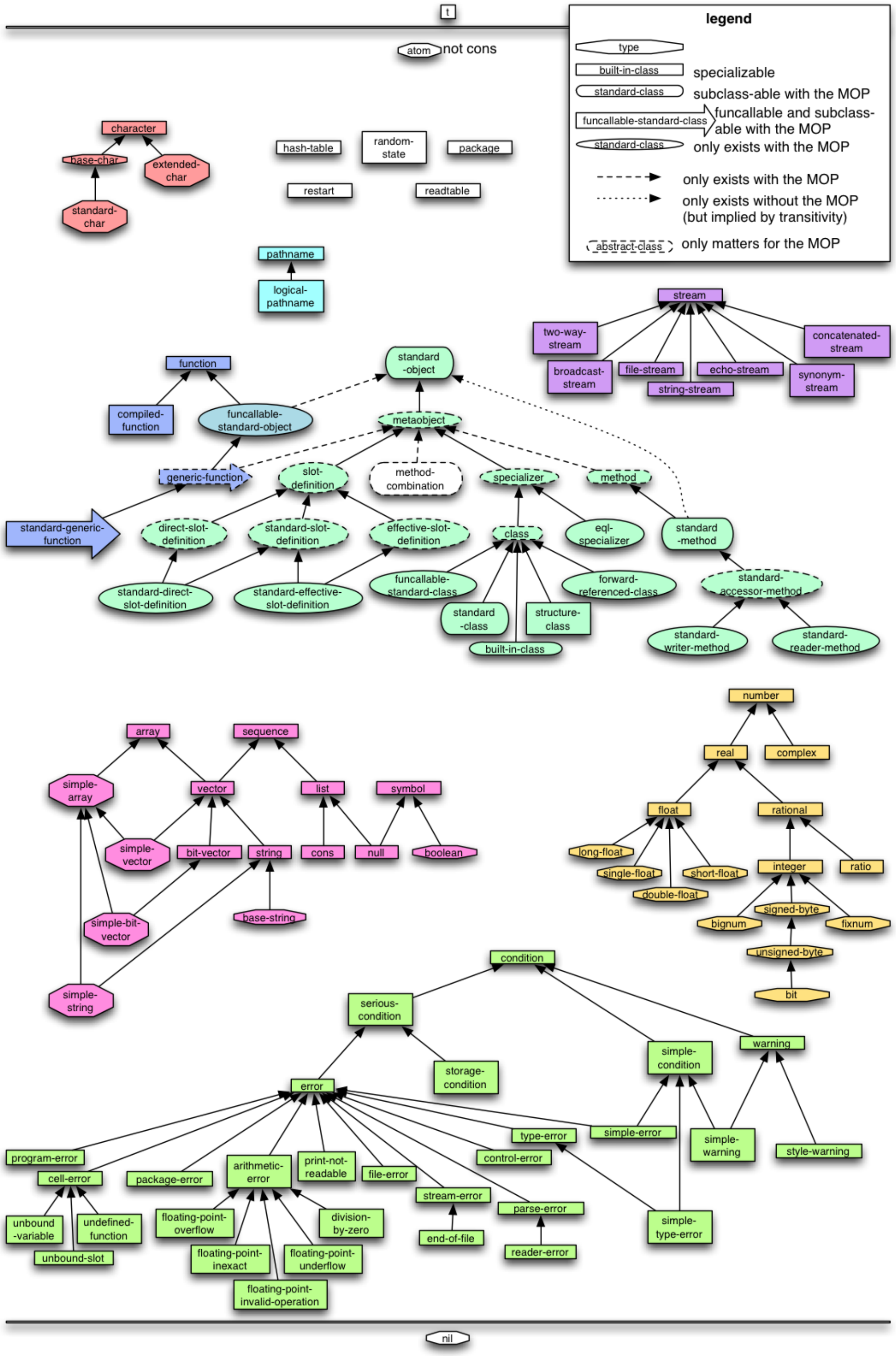
```
    @ variant n - i */
  while (i < n) {
    if (t[i] == v) break;
    i++;
  }
  return i;
}
```

Chapter 11

Types and Signatures

We need to start from a base of the existing types in Common Lisp, eventually providing Axiom combinations or specializations. Common Lisp has these standard type specifier symbols.

Common Lisp Type Hierarchy [Pfei12]



Axiom adds these types:

- Command = String

Chapter 12

COQ nat vs Axiom NNI

COQ's nat domain includes a proof of GCD.

We would like to show an isomorphism between types in Coq and types in Axiom. Having such an isomorphism will make lemmas available and simplify future proofs.

Note that Coq's `nat` domain stops at O (a symbolic 0) as does Axiom's NNI. The Axiom interpreter will promote a subtraction to Integer whereas Coq will not.

COQ's nat domain [COQnat] is

12.0.1 Library Coq.Init.Nat

```
Require Import Notations Logic Datatypes.

Local Open Scope nat_scope.
```

Peano natural numbers, definitions of operations

This file is meant to be used as a whole module, without importing it, leading to qualified definitions (e.g. Nat.pred)

```
Definition t := nat.
```

Constants

```
Definition zero := 0.
Definition one := 1.
Definition two := 2.
```

Basic operations

```
Definition succ := S.

Definition pred n :=
  match n with
    | 0 => n
    | S u => u
  end.

Fixpoint add n m :=
```

```
  match n with
  | 0 => m
  | S p => S (p + m)
  end

where "n + m" := (add n m) : nat_scope.

Definition double n := n + n.

Fixpoint mul n m :=
  match n with
  | 0 => 0
  | S p => m + p * m
  end

where "n * m" := (mul n m) : nat_scope.
```

Note that Axiom's NNI domain will be automatically promoted to Integer when the subtraction result is negative. Coq returns O when this occurs.

```
Truncated subtraction: n-m is 0 if n<=m

Fixpoint sub n m :=
  match n, m with
  | S k, S l => k - l
  | _, _ => n
  end

where "n - m" := (sub n m) : nat_scope.
```

Comparisons

```
Fixpoint eqb n m : bool :=
  match n, m with
    | 0, 0 => true
    | 0, S _ => false
    | S _, 0 => false
    | S n', S m' => eqb n' m'
  end.

Fixpoint leb n m : bool :=
  match n, m with
    | 0, _ => true
    | _, 0 => false
    | S n', S m' => leb n' m'
  end.

Definition ltb n m := leb (S n) m.

Infix "=?" := eqb (at level 70) : nat_scope.
Infix "<=?" := leb (at level 70) : nat_scope.
Infix "<?" := ltb (at level 70) : nat_scope.

Fixpoint compare n m : comparison :=
```

```
  match n, m with
   | 0, 0 => Eq
   | 0, S _ => Lt
   | S _, 0 => Gt
   | S n', S m' => compare n' m'
  end.

Infix "?=" := compare (at level 70) : nat_scope.
```

Minimum, maximum

```
Fixpoint max n m :=
  match n, m with
    | 0, _ => m
    | S n', 0 => n
    | S n', S m' => S (max n' m')
  end.

Fixpoint min n m :=
  match n, m with
    | 0, _ => 0
    | S n', 0 => 0
    | S n', S m' => S (min n' m')
  end.
```

Parity tests

```
Fixpoint even n : bool :=
  match n with
    | 0 => true
    | 1 => false
    | S (S n') => even n'
  end.

Definition odd n := negb (even n).
```

Power

```
Fixpoint pow n m :=
  match m with
    | 0 => 1
    | S m => n * (n^m)
  end

where "n ^ m" := (pow n m) : nat_scope.
```

Euclidean division

```
This division is linear and tail-recursive. In divmod, y is the
predecessor of the actual divisor, and u is y minus the real remainder

Fixpoint divmod x y q u :=
  match x with
    | 0 => (q,u)
```

```
    | S x' => match u with
               | O => divmod x' y (S q) y
               | S u' => divmod x' y q u'
             end
  end.

Definition div x y :=
  match y with
    | O => y
    | S y' => fst (divmod x y' 0 y')
  end.

Definition modulo x y :=
  match y with
    | O => y
    | S y' => y' - snd (divmod x y' 0 y')
  end.

Infix "/" := div : nat_scope.
Infix "mod" := modulo (at level 40, no associativity) : nat_scope.
```

Greatest common divisor

We use Euclid algorithm, which is normally not structural, but Coq is now clever enough to accept this (behind modulo there is a subtraction, which now preserves being a subterm)

```
Fixpoint gcd a b :=
  match a with
   | O => b
   | S a' => gcd (b mod (S a')) (S a')
  end.
```

Square

```
Definition square n := n * n.
```

Square root

The following square root function is linear (and tail-recursive). With Peano representation, we can't do better. For faster algorithm, see Psqrt/Zsqrt/Nsqrt... We search the square root of n = k + p^2 + (q - r) with q = 2p and 0<=r<=q. We start with p=q=r=0, hence looking for the square root of n = k. Then we progressively decrease k and r. When k = S k' and r=0, it means we can use (S p) as new sqrt candidate, since (S k')+p^2+2p = k'+(S p)^2. When k reaches 0, we have found the biggest p^2 square contained in n, hence the square root of n is p.

```
Fixpoint sqrt_iter k p q r :=
  match k with
    | O => p
    | S k' => match r with
               | O => sqrt_iter k' (S p) (S (S q)) (S (S q))
```

```
              | S r' => sqrt_iter k' p q r'
            end
  end.
```

```
Definition sqrt n := sqrt_iter n 0 0 0.
```

Log2

This base-2 logarithm is linear and tail-recursive. In log2_iter, we maintain the logarithm p of the counter q, while r is the distance between q and the next power of 2, more precisely q + S r = 2^(S p) and r<2^p. At each recursive call, q goes up while r goes down. When r is 0, we know that q has almost reached a power of 2, and we increase p at the next call, while resetting r to q. Graphically (numbers are q, stars are r) :

```
          10
         9
        8
       7 *
      6   *
     5     ...
    4
   3 *
  2   *
 1 *   *
0 * *   *
```

We stop when k, the global downward counter reaches 0. At that moment, q is the number we're considering (since k+q is invariant), and p its logarithm.

```
Fixpoint log2_iter k p q r :=
  match k with
    | O => p
    | S k' => match r with
              | O => log2_iter k' (S p) (S q) q
              | S r' => log2_iter k' p (S q) r'
            end
  end.
```

```
Definition log2 n := log2_iter (pred n) 0 1 0.
```

Iterator on natural numbers

```
Definition iter (n:nat) {A} (f:A->A) (x:A) : A :=
 nat_rect (fun _ => A) x (fun _ => f) n.
```

Bitwise operations We provide here some bitwise operations for unary numbers. Some might be really naive, they are just there for fullfiling the same interface as other for natural representations. As soon as binary representations such as NArith are available, it is clearly better to convert to/from them and use their ops.

```
Fixpoint div2 n :=
  match n with
  | 0 => 0
  | S 0 => 0
  | S (S n') => S (div2 n')
  end.

Fixpoint testbit a n : bool :=
 match n with
   | 0 => odd a
   | S n => testbit (div2 a) n
 end.

Definition shiftl a := nat_rect _ a (fun _ => double).
Definition shiftr a := nat_rect _ a (fun _ => div2).

Fixpoint bitwise (op:bool->bool->bool) n a b :=
 match n with
  | 0 => 0
  | S n' =>
    (if op (odd a) (odd b) then 1 else 0) +
    2*(bitwise op n' (div2 a) (div2 b))
 end.

Definition land a b := bitwise andb a a b.
Definition lor a b := bitwise orb (max a b) a b.
Definition ldiff a b := bitwise (fun b b' => andb b (negb b')) a a b.
Definition lxor a b := bitwise xorb (max a b) a b.
```

Chapter 13

Binary Power in COQ by Casteran and Sozeau

From Casteran and Sozeau [Cast16]:

```
(* About integer powers (monomorphic version) *)

Set Implicit Arguments.
Require Import ZArith.
Require Import Div2.
Require Import Program.
Open Scope Z_scope.
```

Let us consider a simple arithmetic operation: raising some integer x to the n-th power, where n is a natural number. The following function definition is a direct translation of the mathematical concept:

```
Fixpoint power (a:Z)(n:nat) :=
  match n with 0%nat => 1
             | S p =>  a * power a p
  end.

Eval vm_compute in power 2 40.
 = 1099511627776 : Z
```

This definition can be considered as a very naive way of programming, since computing x^n requires n multiplications. Nevertheless, this definition is very simple to read, and everyone can admit that it is correct with respect to the mathematical definition. Thus, we can consider it as a *specification*: when we write more efficient but less readable functions for exponentiation, we should be able to prove their correctness by proving in Coq their equivalence with the naive power function.

The following function allows one to compute x^n, with a number of multiplications proportional to $\log_2(n)$:

```
Program
Fixpoint binary_power_mult (acc x:Z) (n:nat) {measure (fun i=>i) n} : Z
  (* acc * (power x n) *) :=
  match n with
```

```
    | 0%nat => acc
    | _ => if Even.even_odd_dec n
           then binary_power_mult acc (x * x) (div2 n)
           else binary_power_mult (acc * x) (x * x) (div2 n)
  end.

Solve Obligations with program_simpl; intros; apply lt_div2; auto with arith.

Definition binary_power (x:Z)(n:nat) := binary_power_mult 1 x n.

Eval vm_compute in binary_power 2 40.
 = 1099511627776 : Z

Goal binary_power 2 234 = power 2 234.
reflexivity.
Qed.
```

We want now to *prove* `binary_power's` correctness, i.e. that this function and the naive `power` function are pointwise equivalent.

Proving this equivalence in Coq may require a lot of work. Thus it is not worth at all writing a proof dedicated only to powers of integers. In fact, the correctness of `binary_power` with respect to `power` holds in any structure composed of an associative binary operation on some domain, that admits a neutral element. For instance, we can compute powers of square matrices using the most efficient of both algorithms.

Thus, let us throw away our previous definition, and try to define them in a more generic framework.

13.1 On Monoids

Definition 2.1 *A monoid is a mathematical structure composed of*

- *a carrier* A
- *a binary, associative operation* $\circ$ *on* A
- *a neutral element* $1 \in A$ *for* $\circ$

Such a mathematical structure can be defined in Coq as a type class. [Soze08]. In the following definition, parameterized by a type A (implicit), a binary operation `dot` and a neutral element `unit`, three fields describe the properties that `dot` and `unit` must satisfy.

```
Class Monoid {A:Type}(dot : A -> A -> A)(one : A) : Prop := {
  dot_assoc : forall x y z:A, dot x (dot y z) = dot (dot x y) z;
  unit_left : forall x, dot one x = x;
  unit_right : forall x, dot x one = x }.
```

Note that other definitions could have been given for representing this mathematical structure.

From an implementational point of view, such a type class is just a record type, i.e. an inductive type with a single constructor `Build_Monoid`

```
Print Monoid.

Record Monoid (A:Type)(dot : A -> A -> A)(one : A) : Prop := Build_Monoid
```

```
  { dot_assoc : forall x y z:A, dot x (dot y z) = dot (dot x y) z;
    one_left : forall x, dot one x = x;
    one_right : forall x, dot x one = x }

For Monoid: Argument A is implicit and maximally inserted
For Build_Monoid: Argument A is implicit
For Monoid: Argument scopes are [type_scope _ _]
For Build_Monoid: Argument scopes are [type_scope _ _ _ _ _]
```

Nevertheless, implementation of type classes by M. Sozeau provides several specific tools — dedicated tactics for instance –, and we advise the reader not to replace the `Class` keyword with `Record` or `Inductive`.

With the command `About`, we can see the polymorphic type of the fields of the class `Monoid`:

```
About one_left

one_left:
forall (A : Type) (dot : A -> A -> A) (one : A),
Monoid dot one -> forall x : A, dot one x = x

Arguments A, dot, one, Monoid are implicit and maximally inserted
Argument scopes are [type_scope _ _ _ _]
one_left is transparent
```

13.1.1 Classes and Instances

Members of a given class are called *instances* of this class. Instances are defined to the Coq system through the `Instance` keyword. Our first example is a definition of the monoid structure on the set $\mathbb{Z}$ of integers, provided with integer multiplication, with 1 as the neutral element. Thus we give these parameters to the `Monoid` class (note that $\mathbb{Z}$ is implicitly given).

```
Instance ZMult : Monoid Zmult 1
```

For this instance to be created, we need to prove that the binary operation `Zmult` is associative and admits 1 as the neutral element. Applying the constructor `Build_Monoid` – for instance with the tactic `split` – generates three subgoals.

```
split.
3 subgoals
 ============================================================
  forall x y z : Z, x * (y * z) = x * y * z

subgoal 2 is:
  forall x : Z, 1 * x = x
subgoal 3 is:
  forall x : Z, x * 1 = x
```

Each subgoal is easily solved by `intros; ring`.

When the proof is finished, we register our instance with a simple `Qed`. Note that we used `Qed` because we consider a class of sort `Prop`. In some cases where instances must store some information constants, ending an instance construction with `Defined` may be necessary.

```
Check Zmult.
ZMult : Monoid Zmult 1
```

We explained on the preceding page why it is better to use the `Class` keyword than `Record` or `Inductive`. For the same reason, the definition of an instance of some class should be written using `Instance` and not `Lemma`, `Theorem`, `Example`, etc. nor `Definition`.

13.1.2 A generic definition of `power`

We are now able to give a definition of the function `power` than can be applied with any instance of class `Monoid`:

A first definition could be

```
Fixpoint power {A:Type}{dot:A->A->A}{one:A}{M: Monoid dot one}
               (a:A)(n:nat) :=
  match n with 0:nat => one
             | S p => dot a (power a p)
  end.

Compute power 2 10.
 = 1024 : Z
```

Happily, we can make the declaration of the three first arguments implicit, by using the `Generalizable Variables` command:

```
Reset power.

Generalizable Variables A dot one.

Fixpoint power `{M: Monoid A dot one}(a:A)(n:nat) :=
  match n with 0%nat => one
             | S p => dot a (power a p)
  end.

Compute power 2 10.
 = 1024 : Z
```

The variables `A dot one` appearing in the binder for `M` are implicitly bound before the binder for `M` and their types are inferred from the `Monoid A dot one` type. This syntactic sugar helps abbreviate bindings for classes with parameters. The resulting internal Coq term is exactly the same as the first definition above.

13.1.3 Instance Resolution

The attentive reader has certainly noticed that in the last computation, the binary operation `Zmult` and the neutral element 1 need not to be given explicitly. The mechanism that allows Coq to infer all the arguments needed by the `power` function to be applied is called *instance resolution*.

In order to understand how it operates, let's have a look at `power`'s type:

```
About power.
power :
forall (A : Type) (dot : A -> A -> A) (one : A),
Monoid dot one -> A -> nat -> A

Arguments A, dot, one, M are implicit and maximally inserted
```

```
Compute power 2 100.
= 1267650600228229401496703205376 : Z

Set Printing Implicit.
Check power 2 100.
@power Z Zmult 1 Zmult 2 100 : Z
Unset Printing Implicit.
```

We see that the *instance* ZMult has been inferred from the type of 2. We are in the simple case where only one monoid of carrier Z has been declared as an instance of the Monoid class.

The implementation of type classes in Coq can retrieve the instance ZMult from the type Z, then filling the arguments ZMult and 1 from ZMult's definition.

13.2 More Monoids

13.2.1 Matrices over some ring

We all know that multiplication of square matrices is associative and admits identity matrices as neutral elements. For simplicity's sake let us restrict our study to 2×2 matrices over some ring.

We first load the Ring library, then open a section with some useful declarations and notations.

```
Require Import Ring.

Section matrices.
  Variables (A:Type)
            (zero one : A)
            (plus mult minus : A -> A -> A)
            (sym : A -> A).
  Notation "0" := zero.
  Notation "1" := one.
  Notation "x + y" := (plus x y).
  Notation "x * y" := (mult x y).

  Variable rt : ring_theory zero one plus mult minus sym (@eq A).

  Add Ring Aring : rt.
```

We can now define a carrier type for 2×2-matrices, as well as matrix multiplication and the identity matrix.

```
Structure M2 : Type := {c00 : A; c01 : A; c10 : A; c11 : A}.

Definition Id2 : M2 := Build_M2 1 0 0 1.

Definition M2_mult (m m':M2) : M2 :=
  Build_M2 (c00 m * c00 m' + c01 m * c10 m')
           (c00 m * c01 m' + c01 m * c11 m')
           (c10 m * c00 m' + c11 m * c10 m')
           (c10 m * c01 m' + c11 m * c11 m').
```

As for multiplication of integers, we can now define an instance of `Monoid` for the type `M2`.

```
Global Instance M2_Monoid : Monoid (M2_mult plus mult) (Id2 0 1).
split.
destruct x; destruct y; destruct z; simpl.
unfold M2_mult. apply M2_eq_intros; simpl; ring.
destruct x; simpl;
unfold M2_mult; apply M2_eq_intros; simpl; ring.
destruct x; simpl;
unfold M2_mult; apply M2_eq_intros; simpl; ring.
Qed.

End matrices.
```

We want now to play with 2×2 matrices over $\mathbb{Z}$. We declare an instance **M2Z** for this purpose, and can use directly the function `power`.

```
Instance M2Z : Monoid _ _ := M2_Monoid Zth.

Compute power (Build_M2 1 1 1 0) 40.
 = {|
     c00 := 165580141;
     c01 := 102334155;
     c10 := 102334155;
     c11 := 63245986 |}
   : M2 Z

Definition fibonacci (n:nat) :=
  C00 (power (Build_M2 1 1 1 0) n).

Compute fibonacci 20.
 = 10946
 :Z
```

13.3 Reasoning within a Type Class

We are now able to consider again the equivalence between two functions for computing powers. Let use define the binary algorithm for any monoid.

First, we define an auxiliary function. We use the `Program` extension to define an efficient version of exponentiation using an accumulator. The function is defined by well-founded recursion on the exponent n.

```
Function binary_power_mult (A:Type) (dot:A->A->A) (one:A)
     (M: @Monoid A dot one) (acc x:A)(n:nat){measure (fun i=>i) n} : A
  (* acc * (x ** n) *) :=
  match n with
    | 0%nat => acc
    | _ => if Even.even_odd_dec n
             then binary_power_mult _ acc (dot x x) (div2 n)
             else binary_power_mult _ (dot acc x) (dot x x) (div2 n)
  end.
intros; apply lt_div2; auto with arith.
intros; apply l2_div2; auto with arith.
Defined.
```

```
Definition binary_power `{M:Monoid} x n := binary_power_mult M one x n.

Compute binary_power 2 100.
 = 1267650600228229401496703205376 : Z
```

13.3.1 The Equivalence Proof

The proof of equivalence between `power` and `binary_power` is quite long, and can be split in several lemmas. Thus, it is useful to open a section, in which we fix some arbitrary monoid M. Such a declaration is made with the `Context` command, which can be considered as a version of `Variables` for declaring arbitrary instances of a given class.

```
Section About_power.

Require Import Arith.
 Context `(M:Monoid A dot one ).
```

It is good practice to define locally some specialized notations and tactics.

```
Ltac monoid_rw :=
    rewrite (@one_left A dot one M) ||
    rewrite (@one_right A dot one M)||
    rewrite (@dot_assoc A dot one M).


Ltac monoid_simpl := repeat monoid_rw.

Local Infix "*" := dot.
Local Infix "**" := power (at level 30, no associativity).
```

13.3.2 Some Useful Lemmas About `power`

We start by proving some well-known equalities about powers in a monoid. Some of these equalities are integrated later in simplification tactics.

```
Lemma power_x_plus : forall x n p, x ** (n + p) =  x ** n *  x ** p.
Proof.
 induction n as [| p IHp];simpl.
   intros;  monoid_simpl;trivial.
  intro q;rewrite (IHp q);  monoid_simpl;trivial.
Qed.

Ltac power_simpl := repeat (monoid_rw || rewrite <- power_x_plus).

Lemma power_commute : forall x n p,
               x ** n * x ** p = x ** p * x ** n.
Proof.
  intros x n p;power_simpl; rewrite (plus_comm n p);trivial.
Qed.

Lemma power_commute_with_x : forall x n ,
        x * x ** n = x ** n * x.
Proof.
```

```
  induction n;simpl;power_simpl;trivial.
  repeat rewrite <- (@dot_assoc A dot one M); rewrite IHn; trivial.
Qed.

Lemma power_of_power : forall x n p,  (x ** n) ** p = x ** (p * n).
Proof.
   induction p;simpl;[| rewrite power_x_plus; rewrite IHp]; trivial.
Qed.

Lemma power_S : forall x n, x *  x ** n = x ** S n.
Proof. intros;simpl;auto. Qed.

Lemma sqr : forall x, x ** 2 =  x * x.
Proof.
 simpl;intros;monoid_simpl;trivial.
Qed.

Ltac factorize := repeat (
                rewrite <- power_commute_with_x ||
                rewrite  <- power_x_plus  ||
                rewrite <- sqr ||
                rewrite power_S ||
                rewrite power_of_power).

Lemma power_of_square : forall x n, (x * x) ** n = x ** n * x ** n.
  induction n;simpl;monoid_simpl;trivial.
  repeat rewrite dot_assoc;rewrite IHn; repeat rewrite dot_assoc.
 factorize; simpl;trivial.
Qed.
```

13.3.3 Final Steps

We are now able to prove that the auxiliary function binary_power_mult satisfies its intuitive meaning. The proof uses well-founded induction and the lemmas proven in the previous section.

```
Lemma binary_power_mult_ok :
  forall n a x,  binary_power_mult a x n = a * x ** n.
Proof.
  intro n; pattern n;apply lt_wf_ind.
  clear n; intros n Hn;   destruct n.
   intros;simpl; monoid_simpl;  trivial.
  intros; rewrite binary_power_mult_equation.
     destruct (Even.even_odd_dec (S n)).
   rewrite Hn.   rewrite power_of_square;  factorize.
   pattern (S n) at 3;replace (S n) with (div2 (S n) + div2 (S n))%nat;auto.
   generalize (even_double _ e);simpl;auto.
   apply lt_div2;auto with arith.
   rewrite Hn.
  rewrite power_of_square ; factorize.
  pattern (S n) at 3;replace (S n) with (S (div2 (S n) + div2 (S n)))%nat;auto.

  rewrite <- dot_assoc; factorize;auto.
```

```
  generalize (odd_double _ o);intro H;auto.
  apply lt_div2;auto with arith.
Qed.
```

Then the main theorem follows immediately:

```
Lemma binary_power_ok : forall (x:A) (n:nat), binary_power x n = x ** n.
Proof.
  intros n x;unfold binary_power;rewrite binary_power_mult_ok;
  monoid_simpl;auto.
Qed.
```

13.3.4 Discharging the Context

It is time to close the section we opened for writing our proof of equivalence. The theorem `binary_power_ok` is now provided with a universal quantification over all the parameters of any monoid.

```
End About_power.

About binary_power_ok.
binary_power_ok :
forall (A : Type) (dot : A -> A -> A) (one : A) (M : Monoid dot one)
  (x : A) (n : nat), binary_power x n = power x n

Arguments A, dot, one M are implicit and maximally inserted
Argument scopes are [type_scope _ _ _ _ nat_scope]
binary_power_ok is opaque
Expands to Constant Top.binary_power_ok

Check binary_power_ok 2 20.
binary_power_ok 2 20
  : binary_power 2 20 = power 2 20

Let Mfib := Build_M2  1 1 1 0.

Check  binary_power_ok Mfib 56.
binary_power_ok Mfib 56
  : binary_power Mfib 56 = power Mfib 56
```

13.3.5 Subclasses

We could prove many useful equalities in the section `about_power`. Nevertheless, we couldn't prove the equality $(xy)^n = x^n y^n$ because it is false in general – consider for instance the free monoid of strings, or simply matrix multiplication. But this equality holds in every commutative (a.k.a Abelian) monoid.

Thus we say that Abelian monoids form a *subclass* of the class of monoids, and prove this equality in a context declaring an arbitrary instance of this subclass.

Structurally, we parameterize the new class `Abelian_Monoid` by an arbitrary instance M of `Monoid`, and add a new field stating the commutativity of `dot`. Please keep in mind that we declared `A`, `dot`, and `one` as *generalizable variables*, hence we can use the backquote symbol here.

```
Class Abelian_Monoid `(M:Monoid A dot one) := {
  dot_comm : forall x y, dot x y = dot y x}.
```

A quick look at the representation of *Abelian_Monoid* as a record type helps us understand how this class is implemented.

```
Print Abelian_Monoid.
Record Abelian_Monoid (A : Type) (dot : A -> A -> A)
  (one : A) (M : Monoid dot one) : Prop := Build_Abelian_Monoid
  {dot_comm : forall x y : A, dot x y = dot y x }

For Abelian_Monoid: Arguments A, dot, one are implicit and maximally inserted
For Build_Abelian_Monoid: Arguments A, dot, one are implicit
For Abelian_Monoid: Arguemnt scopes are [type_scope _ _ _]
For Build_Abelian_Monoid: Argument scopes are [type_scope _ _ _ _]
```

For building an instance of `Abelian_Monoid` we can start from `ZMult`, the monoid on `Z`, adding a proof that integer multiplication is commutative.

```
Instance ZMult_Abelian : Abelian_Monoid ZMult.
split.
 exact Zmult_comm.
Qed.
```

We can now prove our equality by building an appropriate context. Note that we can specify just the parameters of the monoid here in the binder of the Abelian monoid, an instance of monoid on those same parameters is automatically generalized. Superclass parameters are automatically generalized inside quote binders. Again, this is simply syntactic sugar.

```
Section Power_of_dot.
 Context `{M: Monoid A} {AM:Abelian_Monoid M}.

Theorem power_of_mult : forall n x y,
    power (dot x y)  n =  dot (power x  n) (power y n).
Proof.
 induction n;simpl.
 rewrite one_left;auto.
 intros; rewrite IHn; repeat rewrite dot_assoc.
  rewrite <- (dot_assoc  x y (power x n)); rewrite (dot_comm y (power x n)).
 repeat rewrite dot_assoc;trivial.
Qed.

End Power_of_dot.

Check power_of_mult 3 4 5.
power_of_mult 3 4 5
     : power (4 * 5) 3 = power 4 3 * power 5 3
```

Chapter 14

Proof Tower Layer: C11 using CH_2O

From Krebbers [Kreb17]

Module example_gcd

```
Require Import String axiomatic_simple.

Section gcd.
Context `{EnvSpec K}.

Hint Extern 10 (Some Readable ⊆ _) ⇒ transitivity (Some Writable).
Hint Extern 0 (perm_locked _ = _) ⇒
  apply perm_Readable_locked; auto : typeclass_instances.


Hint Resolve ax_load' ax_var' assert_memext_l' assert_eval_int_cast_self'
  assert_memext_r' assert_and_l assert_singleton_eval assert_int_typed_eval
  assert_eval_singleton_r assert_eval_singleton_l assert_and_intro : exec.

Ltac exec :=
  repeat match goal with A := _ : assert _ ⊢ _ ⇒ progress unfold A end;
  simpl; eauto 20 with exec.

Definition gcd_stmt : stmt K :=
  "l" :; if{load (var 1)} local{uintT} (
    !(var 2 ::= (
      var 0 ::= load (var 1) @{ArithOp ModOp} load (var 2),,
      var 1 ::= load (var 2),,
      load (var 0)));;
    goto "l"
  ) else skip.

Lemma gcd_typed : (∅,∅,[uintT%T;uintT%T]) ⊢ gcd_stmt : (false,None).
Proof.
Lemma gcd_correct Γ δ R J T C y z μ1 γ1 μ2 γ2 :
  sep_valid γ1 → Some Writable ⊆ perm_kind γ1 →
  sep_valid γ2 → Some Writable ⊆ perm_kind γ2 →
```

J "l"%string ≡ {Γ,δ} (∃ y' z',
⌜ Z.gcd y' z' = Z.gcd y z ⌝%Z
var 0 ↦ {μ1,γ1} #intV{uintT} y' : uintT
var 1 ↦ {μ2,γ2} #intV{uintT} z' : uintT)%A →
Γ δ R J T C $\models_s$
{{ var 0 ↦ {μ1,γ1} #intV{uintT} y : uintT
var 1 ↦ {μ2,γ2} #intV{uintT} z : uintT }}
gcd_stmt
{{ var 0 ↦ {μ1,γ1} #intV{uintT} (Z.gcd y z) : uintT
var 1 ↦ {μ2,γ2} #intV{uintT} 0 : uintT }}.
Proof.
End gcd.

Chapter 15

Other Ideas to Explore

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.

Spitters [Spit11] Type Classes for Mathematics in Coq. Also see `http://www.eelis.net/research/math-classes/`

Mahboubi [Mahb16] Mathematical Components. This book contains a proof of the Euclidean algorithm using COQ.

Aczel [Acze13] Homotopy Type Theory

Santas [Sant95] A Type System for Computer Algebra

Homann [Homa94] algorithm schemata

Name: gcd(?a,?b)=?g
Signature: ?A $\times$?A $\rightarrow$?A
Constraints: (?A, EuclideanRing)
Definition: $(?g|?a) \wedge (?g|?b) \wedge (\forall c \in ?A : (c|?a) \wedge (c|?b) \Rightarrow (c|?g))$
Theorems:
gcd(u,v) = gcd(v,u)
gcd(u,v) = gcd(v, u mod v)
gcd(u,0) = u

Appendix A

The Global Environment

Let S be a set. Let $\circ$ be a binary operation. Let $+$ be an additive operation. Let $*$ be a multiplicative operation.

Axiom 1 (Magma) *A* **Magma** *is the set S with a* **closed binary operation** $S \circ S \to S$ *such that*

$$\forall a, b \in S \Rightarrow a \circ b \in S$$

.

Axiom 2 (Semigroup) *A* **Semigroup** *is a* **Magma** *with the operation $\circ$ that is* **associative** *such that*

$$\forall a, b, c \in S \Rightarrow (a \circ b) \circ c = a \circ (b \circ c)$$

Axiom 3 (Abelian Semigroup) *An* **Abelian Semigroup** *is a* **Semigroup** *with the operation $\circ$ that is* **commutative** *such that*

$$\forall a, b \in S \Rightarrow a \circ b = b \circ a$$

Axiom 4 (Monoid) *A* **Monoid** *is a* **Semigroup** *with an* **identity element** $e \in S$ *such that*

$$\forall a \in S \Rightarrow e \circ a = a \circ e = a$$

Axiom 5 (Group) *A* **Group** *is a* **Monoid** *with an* **inverse element** $b \in S$ *and an identity element $i \in S$ such that*

$$\forall a \in S\ \exists b \in S \Rightarrow a \circ b = b \circ a = i$$

Axiom 6 (Group Unique Identity) *A* **Group** *has a* **unique identity element** $e \in S$ *such that*

$$\exists e \wedge \forall a, b \in S \wedge a \neq e \wedge b \neq e \Rightarrow a \circ b \neq e$$

Axiom 7 (Group Unique Inverse) *A* **Group** *has a* **unique inverse element** $i \in S$ *such that*

$$\exists i \wedge \forall a, b \in S \wedge a \neq i \wedge b \neq i \Rightarrow a \circ b \neq i$$

Axiom 8 (Group Right Quotient) *A* **Group** *has a* **Right Quotient** *(right division) such that*

$$x \circ a = b \Rightarrow x \circ a \circ a^{-1} = b \circ a^{-1} \Rightarrow x = b \circ a^{-1}$$

Axiom 9 (Group Left Quotient) *A* **Group** *has a* **Left Quotient** *(left division) such that*

$$a \circ x = b \Rightarrow a^{-1} \circ a \circ x = a^{-1} \circ b \Rightarrow x = a^{-1} \circ b$$

Axiom 10 (Abelian Group) *An* **Abelian Group** *is a* **Group** *with the operation* $\circ$ *that is* **commutative** *such that*

$$\forall a, b \in S \Rightarrow a \circ b = b \circ a$$

Axiom 11 (Abelian Group Quotient) *An* **Abelian Group** *has a* **Quotient** *(division) such that*

$$a^{-1} \circ a \circ x = a \circ a^{-1} \circ x$$

Axiom 12 (Euclidean Domain) *Let R be an integral domain. Let f be a function from $R\backslash\{0\}$ to the NonNegativeInteger domain. If a and b are in R and b is nonzero, then there are q and r in R such that $a = bq + r$ and either $r = 0$ or $f(r) < f(b)$*

Appendix B

Related work

B.1 Overview of related work

B.1.1 Adams [Adam01]

B.1.2 Ballarin [Ball95]

B.1.3 Berger and Schwichtenberg [Berg95]

The Greatest Common Divisor: A Case Study for Program Extraction from Classical Proofs.

Theorem

$$\forall a_1, a_2(0 < a_2 \to \exists k_1, k_2(abs(k_1a_1 - k_2a_2)|a_1 \wedge abs(k_1a_1 - k_2a_2)|a_2 \wedge 0 < abs(k_1a_1 - k_2a_2)))$$

Proof Let a_1, a_2 be given and assume $0 < a_2$. The ideal (a_1, a_2) generated from a_1, a_2 has a least positive element c, since $0 < a_2$. This element has a representation $c = abs(k_1a_1 - k_2a_2)$ with $k_1, k_2 \in \mathbb{N}$. It is a common divisor of a_1 and a_2 since otherwise the remainder $f(a_i, c)$ would be a smaller positive element of the ideal. The number $c \in (a_1, a_2)$ dividing a_1 and a_2 is the greatest common divisor since any common divisor of a_1 and a_2 must also be a divisor of c.

```
(lambda (a1)
 (lambda (a2)
   ((((((nat-rec-at '(arrow nat (arrow nat (start nat nat))))
        (lambda (k1) (lambda (k2) (cons n000 n000))))
       (lambda (n)
        (lambda (w)
         (lambda (k1)
          (lambda (2)
           ((((if-at '(star nat nat))
              ((<-strict-nat 0) r2))
                ((w L21) L22))
                ((((if-at '(star nat nat))
                  ((<-strict-nat 0) r1))
```

```
                ((w L11 L12))
               (cons k1 k2))))))))
         ((plus-nat a2) 1))
      0)
    1)))
```

Here we have manually introduced r1,r2,L11,L12,L21,L22 for somewhat lengthy terms corresponding to our abbreviations r_1, $\overline{\to l_1}$. The unbound variable n000 appearing in the base case is a dummy variable used by the system when it is asked to produce a realizing term for the instance $\perp \to \exists k A(k)$ of ex-falso-quodlibet. In our case, when the existential quantifier is of type nat one might as well pick the constant 0 (as we did in the text).

B.1.4 Cardelli [Card85]

Cardelli states that **coercions** are a form of ad-hoc polymorphism. Axiom allows ML-style type inference in the interpreter. Declaring a function

```
f(n) == n + 1
```

without specifying types is possible. The function types are inferred when used and a type-specialized version of the function is compiled. If called with different argument types it will compile a new type-specialized version of the function as needed. Raises the distinction between **equivalence** and **inclusion** as it occurs in subtypes (p483). The FUN language defines **Quantified Types** as

QuantifiedType ::=

∀ A, Type |	Universal Quantification
∃ A. Type |	Existential Quantification
∀ A ⊆ Type. Type | ∃ A ⊆ Type. Type	Bounded Quantification

See p516 for the Classification of Type Systems diagram.

B.1.5 Clarke [Clar91]

Clarke shows several proofs

B.1.6 Crocker [Croc14]

Primary specification constructs

pre(expression-list)	Declares preconditions
post(expression-list)	Declares postconditions
returns(expression)	Declares the value returned by a function. Equivalent to **post(result)** == *expression* except that recursion is permitted in *expression*
assert(expression-list)	Asserts conditions
invariant(expression-list)	Used in class declarations to declare class invariants, and in **typedef** declarations to declare constraints
keep(expression-list)	Declares loop invariants
decrease(expression-list)	Declares loop variant or recursion variant expressions
writes(lvalue-expression-list)	Declares what non-local variables the function modifies. If a function is declared without a writes-clause, then a default writes-clause is constructed based on the signature of the function
assumes(expression-list)	Declares predicates to be assumed without proof
ghost(expression-list)	Declares ghost variables, functions parameters, etc.

Additional specification expressions

exists *identifier* **in** *expression* :- *predicate*	Existential quantification over the elements of *expression* which must be an array or an abstract collection type
exists *type identifier* := *predicate*	Existential quantification over all values of type
forall *identifier* **in** *expression* :- *predicate*	Universal quantification over the elements of *expression*, which must be an array or an abstract collection type
forall *type identifier* :- *predicate*	Universal quantification over all values of type
for *identifier* **in** *expression1* **yield** *expression2*	Applies the mapping function *expression2* to each element of collection *expression1*, yielding a new collection
those *identifer* **in** *expression1* :- *predicate*	Selects those elements of collection *expression1* for which *predicate* is true
that *identifier* **in** *expression1* :- *predicate*	Selects the single element of the collection *expression1* for which *predicate* is true
expression1 **in** *expression2*	Shorthand for **exists** *id* **in** *expression2* :- *id* == *expression1*, where *id* is a new identifier
expression **holds** *member*	*expression* must have union type, and *member* must be a member of that type. Yields true if and only if the value of *expression* was defined by assignment or initialization through *member*
disjoint(*lvalue-expression-list*)	Yields true if and only if no two objects in the expression list have overlapping storage. Typically used in preconditions to state that parameters passed by pointer or reference refer to distinct objects
operator **over** *expression*	Left-fold *operator* over collection *expression*. Used to express e.g. summation of the elements of an array
old*(expreesion)*	When used in a postcondition, this referes to the value of *expression* when the function was entered. When used in a loop invariant, it refers to the value of *expression* just before the first iteration of the loop.

B.1.7 Davenport [Dave02]

B.1.8 Davis [Davi09]

Davis creates a verified proof checker by creating a simple, obviously correct "level 1" checker based on a few rules. A "level 2" checker is implemented and verified by level 1. This bootsrap process continues up to level 11 which has significant power. Every level 11 proof can be reduced to a level 1 proof, giving a solid foundation.

B.1.9 Filliatre [Fill03]

A formal method to establish software correctness can involve several steps. The first one is the *specification*. A second one is a method to generate some *proof obligations*. And a third one is a framework to *establish their validity*.

Type theory identifies types with propositions and terms with proofs, through the widely known Curry-Howard isomorphism. There is no real difference between the usual first-order objects of the mathematical discourse – such as naturals, sets, and so forth – and the proof objects. The natural 2 is a first-order object of type `nat`, and a proof that 2 is even is a first-order object of type *even*(2). One can define a function f taking as arguments a natural n and a proof that n is even, and its type would be something like $\forall n : \texttt{nat}.even(n) \rightarrow \tau$. Such a function represents a *partial function* on naturals, where the proof of *even(n)* may be seen as a *precondition*. Similarly, one can define a function returning a proof term. For instance, the function f could return a natural p and a proof that $n = 2 \times p$. Finally, the type of f will look like

$$\forall n : \texttt{nat}.even(n) \rightarrow \exists p : \texttt{nat}.n = 2 \times p$$

where the proof of $n = 2 \times p$ may be seen as a *postcondition*. More generally, a type of the form

$$forallx : nat.P(x) \rightarrow \exists y : nat.Q(x, y)$$

is the type of a function with a precondition P and a postcondition Q. Building a term of this type is exactly like building a function together with a proof of its correctness, and consequently type theory appears as naturally suited for the proof of purely functional programs.

We propose an interpretation of the Hoare triple $\{P\}e\{Q\}$ as a proof of the above proposition and then define a systematic construction of this proof from a given annotated program, where the lacking proof terms are the so-called proof obligations.

The Coq **Correctness** tactic takes an annotated program as argument and generates a set of goals, which are logical propositions to be proved by the user. Given an annotated program e, the tactic **Correctness** applies the following steps:

1. It determines the type of computation of κ of e by the typing algorithm
2. proposition $\hat{\kappa}$ is computed and declared as a goal
3. The partial proof term $\hat{e}$ is computed following Definition 8 and is given to the proof engine, using the `Refine` tactic developed on purpose, and each hole in $\hat{e}$ leads to a subgoal
4. Once the proofs are completed, the program is added to the environment and may be used in other programs

Some features have been added to simplify the specification of programs – mainly, the possibility of inserting labels in the programs and referring in annotations to the value that a reference had at the program points corresponding to those labels. In particular, a loop invariant may mention the values of the references at some point before the loop using such a label.

B.1.10 Frege [Freg1891]

Function and Concept seminal paper

Frühwirht [Frue91] details optimistic type systems for logic programs.

B.1.11 Harrison [Harr98, p13]

There are several examples of computer algebra results which may be checked relatively easily:

- factoring polynomials (or numbers)
- finding GCDs of polynomials (or numbers)
- solving equations (algebraic, simultaneous, differential,...)
- finding antiderviatives
- finding closed forms for summations

In most cases the certificate is simply the answer. An exception is the GCD, where a slightly more elaborate certificate is better for our purposes. If we ask to find the GCD of $x^2 - 1$ and $x^5 + 1$ using the gcd function, for example, the respons is $x + 1$. How can this result be checked? It's certainly straightforward to check that this is a common divisor. If we don't want to code polynomial division ourselves in HOL, we can call the `divide` function, and then simply verify the quotient as above. But how can we prove that $x + 1$ is the *greatest* common divisor[1] At first sight, there is no easy way, short of replicating something like the Euclidean algorithm inside the logic (although that isn't really a difficult prospect).

However, a variant GCD algorithm, called *gcdex* will, given polynomials p and q, produce not just the GCD d, but also two other polynomials r and s such that $d = pr + qs$. (Indeed, the coefficients in this sort of Bezout identity follow easily from the Euclidean GCD algorithm.) For example, applied to $x^2 - 1$ and $x^5 + 1$ we get the following equation:

$$(-x^3 - x)(x^2 - 1) + 1(x^5 + 1) = x + 1$$

This again can be checked easily, and from that, the fact that $x + 1$ is the *greatest* common divisor follows by an easily proved theorem, since obviously any common factor of $x^2 - 1$ and $x^5 + 1$ must, by the above equation, divide $x + 1$ too. So here, given a certificate slightly more elaborate than simply the answer, easy and efficient checking is possible.

[1] The use of "greatest" is a misnomer: in a general ring we say that a is the GCD of b and c iff it is a common divisor, and any other common divisor of b and c divides a. For example, both 2 and -2 are GCDs of 8 and 10 over $\mathbb{Z}$.

B.1.12 Hoare [Hoar87]

Let us suppose first that they agree to confine attention to positive whole numbers (exclusing zero). The required relationship between the parameters (x,y) and the result (z) may be formalized as follows:

```
D1.1  z divides x
D1.2  z divides y
D1.3  z is the greatest of the set of numbers satisfying both conditions
D1.4  "p divides q" means "there exists a positive whole number w
      such that pw=q"
D1.5  "p is the greatest member of the set S" means "p is in S, and no
      member of S is strictly greater than p"
```

We need to check that for every pair of positive numbers x and y there exists a number z with the properties specified in D1.3. A proof of this has three steps.

```
P1.1 The number one is a divisor of every number. So it is a common
     divisor of every pair of numbers. This shows that the set of
     common divisors of two numbers is non-empty.
P1.2 Each number is its own greatest divisor, so every set of
     divisors is finite. The common subset of any two finite sets is
     also finite. So the set of common divisors of two nubers is
     both finite and non-empty.
P1.3 Every finite non-empty set of integers has a greatest member.
     So the maximum used to define the greatest common divisor
     always exists.
```

Here is an idealized logic program to compute the greatest common divisor of two positive integers. To help in checking its correctness, it has been designed to preserve as far as possible the structure and clarity of the original requirements. We assume that “isproduct” and “differsfrom” are available as built-in predicates on positive integers.

```
L2.1 isdivisor(x,z) if there exists a w not greater than x such that
     isproduct(z,w,x)
L2.2 iscommutative(x,y,z) if isdivisor(x,z) and isdivisor(y,z)
L2.3 isgcd(x,y,z) if iscommondiv(x,y.z) and for all w from z to x
     isnotcommondiv(x,y,z)
L2.4 isnotcommondiv(x,y,z) if isnotdiv(x,z) or isnotdiv(y,z)
L2.5 isnotdiv(x,z) if for all w from 1 to x isnotproduct(z,w,x)
L2.6 isnotproduct(z,w,x) if isproduct(z,w,y) and differsfrom(y,x)
```

This program is a great deal more complicated than the requirements specification in the preious section. The obvious reason is that the absence of negation in the programming language requires explicit programming of a search through all possibilities before a negative answer is given. In order to ensure termination a finite range for each search has to be specified, and setting this limit requires knowledge of the application domain. For example, in L2.3 we rely on the fact that the common divisor of two numbers cannot exceed either number.

When restricted from using disjunction and negation the algebraic equations have to be derived as needed by mathematical reasoning from the whole of the original specification. For gcd we see

```
L3.1 The greatest divisor of x is x. So the greatest common divisor
     of x and x is also x.
        x = gcd(x,x) for all x
```

```
L3.2 If z divides x and y, it also divides x+y. So every common
     divisor of x and y is also a common divisor of x+y and y.
     Similarly, every common divisor of x+y and y is also a common
     divisor of x and y. So the greatest of these identical sets
     of common divisors are the same.
        gcd(x,y) = gcd(x+y,y) for all x,y
L3.3 Every common divisor of x and y is also a common divisor of
     y and x.
        gcd(x,y) = gcd(y,x) for all x,y
```

But are the laws a *complete* specification, in the sense that there is aonly one function satisfying them? Or do we need to look for more laws? A proof of completeness has to show that for any given positive numerals p and q there is a numeral r such that the equation

$$r = gcd(p, q)$$

can be proved solely from the algebraic specification and the previously known laws of arithmetic.

This can be shown by mathematical induction: We assume the result for all p and q strictly less than N, and prove it for all p and q less than or equal to N. For such numbers, four cases can be distinguished.

1. Both p and q are strictly less than N. In this case, what we have to prove is the same as the induction hypothesis, which may be assumed without proof.
2. Both p and q are equal to N. Then the result

   $$N = gcd(p, q)$$

   is proved immediately by law L3.1
3. $p = N$ and $q < N$. It follows that $p - q$ is positive and less than N. By the induction hypothesis, there is an r such that

   $$r = gcd(p - q, q)$$

   is deducible from the algebraic laws. One application of L3.2 then gives

   $$r = gcd(p - q) + q, q)$$

   which by the laws of arithmetic leads to the required conclusion

   $$r = gcd(p, q)$$

4. $p < n$ and $q = N$. Then there is an r such that

   $$r = gcd(q, p)$$

   is provable in the same way as in case (3) described above. One application of L3.3 then gives

   $$r = gcd(p, q)$$

   That concludes the proof that the algebraic specification is complete.

Clearly there is no structural correspondence between the three clauses of the algebraic specification and the five clauses expressing the original requirement. As a result, some mathematical ingenuity and labour has been needed to prove that the two orthogonal specifications describe (and completely describe) the same function. This labor could be avoided by simply leaving out the original formalization of requirements in the general notations of mathematics, and by starting instead within the more restricted equational framework of algebra.

But this would be a mistake. The purpose of the specification is to tell the user of a subroutine the properties of the result it produces, and to do so in a manner conducive to the wider objectives of the program as a whole. Clearly, the user of a subroutine to compute the greatest common divisor will be very directly interested in the fact that the result of every subroutine call divides each of its two arguments exactly. But the algebraic law tells us only that the same result have been obtained if the two arguments had been permuted (L3.3) or added together (L3.2) before the call. These facts by themselves seem a lot less directly useful.

It would also be a mistake to regard the different specification, the abstract one and the algebraic one, as rivals or even as alternatives. They are both needed; they are essentially complementary, and the can be used for different purposes.

Here is a functional program to compute the greatest common divisor of positive integers.

F4.1	$gcd(x, y) = x$	if $x = y$
F4.2	$gcd(x, y) = gcd(x - y, y)$	if $x > y$
F4.3	$gcd(x, y) = gcd(y, x)$	if $x < y$

To compute the greatest common divisor of 10 and 6:

gcd(10,6)	=	gcd(4,6)	by F4.2
	=	gcd(6,4)	by F4.3
	=	gcd(2,4)	by F4.2
	=	gcd(4,2)	by F4.3
	=	gcd(2,2)	by F4.2
	=	2	by F4.1

However, the responsibility for controlling this goal-directed behavior is placed upon the programmer, who has to prove that there is no infinite chain of substitutions. For example, as an algebraic formula

$$gcd(x, y) = gcd(y, x)$$

is quite correct, but if this is executed as part of a functional program, it leads to an infinite chain of substitutions. In the program shown above, this cycle is broken by ensuring that the dangerous substitution is made only when y is strictly greater than x.

We can optimize the search for the greatest common divisor algebraically.

- L5.1 If z divides x, the $2z$ divides $2x$. So if z is the greatest common divisor of x and y, then $2z$ is a common divisor of $2x$ and $2y$. It is therefore not greater than their greatest common divisor

  $$2gcd(x, y) < gcd(2x, 2y)$$

  Conversely, if z is the greatest common divisor of $2x$ and $2y$, then z is even and $z/2$ is a common divisor of x and y.

  $$gcd(2x, y2)/2 \leq gcd(x, y)$$

From these two inequalities ti follows that

$$2gcd(x, y) = gcd(2x, 2y)$$

- L5.2 All divisors of an odd number are odd, and if an odd number divides $2x$ is also divides x. If y is odd, the greatest common divisor of $2x$ and y is odd, so it is also a common divisor of x and y

  $$gcd(2x, y) \leq gcd(x, y) \quad \text{if } y \text{ is odd}$$

  Conversely, every divisor of x divides $2x$

  $$gcd(x, y) \leq gcd(2x, y)$$

  From these two inequalities it follows that

  $$gcd(2x, y) = gcd(x, y) \quad \text{if } y \text{ is odd}$$

- L5.3 If both x and y are odd, and x is greater than y, it follows that $x - y$ is positive and even. So under these conditions

  $$gcd(x, y) = gcd((x - y)/2, x)$$

When these equations are coded as a functional program, it becomes clear that the number of operations required when the argument of size 2^N has been reduced to about N.

A Procedural program has some assertions, generally specified as preconditions, invariants, and postconditions.

- P6.1 $x > 0 \wedge y > 0$ as a precondition
- P6.2 $Z = gcd(x, y)$ as a postcondition
- P6.3 $s^N gcd(X, Y) = gcd(x, y)$. The task of the first part is to make P6.3 true on termination. That is easily accomplished by just one multiple assignment

  $$N, Z, Y := 0, x, y$$

  which can be proven by substitution

  $$2^0 gcd(x, y) = gcd(x, y)$$

- P6.4 We can show that

  $$2^N Z = gcd(x, y)$$

  This would be obviously true if N were already zero. If N is non-zero, it can be made closer to zero by subtracting one. But ath would amek P6.4 false, and therefore useless. Fortunately, the truth of P6.4 can easily be restored if every subtraction of one from N is accompanied by a doubling of Z. This can be proven by

  $$n > 0 \wedge 2^N Z = gcd(x, y) \rightarrow 2^{N-1}(2Z) = gcd(x, y)$$

  Since termination is obvious, we have proved the correctness of the loop

  $$\textbf{while } N > 0 \textbf{ do } N, Z := N - 1, 2X$$

  On termination of this loop, the value of N is zero and P6.4 is still true. Consequently, the postcondition of the whole program has been established.

Having completed the first and last of the three tasks, the time has come to confess that the middle task is the most difficult. Its precondition is P6.3 and its postcondition is P6.4. The task can be split into four subtasks, in accordance with the following series of intermediate assertions.

P6.3	$\wedge$	(X odd $\vee$ Y odd)
P6.3	$\wedge$	(Y add)
P6.3	$\wedge$	(Y odd) $\wedge$ X=Y

B.1.13 Jenks [Jenk84b]

Overview of Scratchpad.

B.1.14 Kifer [Kife91]

Typed Predicate Calculus giving declarative meaning to logic programs with type declarations and type inference.

B.1.15 Meshveliani [Mesh16a]

Prejudice 1: "Proof by contradiction is not possible in constructive mathematics"

In fact: *it is possible* – when the relation has a **decision algorithm**.

Example: In most domains in computer algebra the equality relation has a decision algorithm `_=?_`. Respecitvely, a program of the kind.

$$\text{case x } =? \text{ y of } \backslash\{(yesx \approx y) \to \ldots; (\text{ no } x \neq y) \to \ldots\}$$

actually applies the *excluded third law* to this relation.

Prejudice 2: "Programs in the verified programming tools (like Coq, Agda) do not provide a proof itself, instead they provide an algorithm to build a proof witness for each concrete data".

I claim: *they also provide a proof in its ordinary meaning* (this is so in Agda, and I expect, the same is with Coq).

B.1.16 [Neup13]

The case study was motivated by a master's thesis at RISC Linz, which implemented a CA algorithm for the greatest common divisor of multivariate polynomials in SML [Mein13].

B.1.17 Smolka [Smol89a]

details the foundations for relational logic programming with polymorphically order-sorted data types.

B.1.18 Strub, Pierre Yves

```
Formal Proofs and Decision Procedures  (Strub, Pierre Yves)
```

```
htps://www.youtube.com/watch?v=Yg0oDNIT8A8
year = "2016"

Prop is a type containing other types
The inhabitants of Prop are proof types or propositions
If A:Prop and B:Prop are proof types then A->B is a proof type (implication)
A proof type is valid if there exists a program of that type
(fun (x:A) => x) is a proof of (A -> A)

Elimination of the \forall connector
   \forall x,P
   --------
   P{x \mapsto t}
 We want to express a proof of (\forall x,P) by a function taking a t
 and returning a proof of P(t)
 The result type depends on the input
    Dependent typ: type that depends on a value

Arithmetic (P. Cregut)
Real Arithmetic (L Pottier)
First Order Logic (P Corbineau)
Polynomial Systems (B Barras, B Gregoire, A Mahboubi)
External Oracles (N Ayache JC Filliatre)

two terms are computationally equal are considered identical

Fact fact1 A B C : (A -> B -> C) -> (A -> B) -> A -> C
fact1 = fun (A B C : Type) (HABC : A -> B -> C) (HAB : A -> B) (HA : A) =>
HABC HA (HAB HA) : forall A B C : Type, (A -> B -> C) -> (A -> B) -> A -> C

Fact fact2 A B C : (A -> B -> C) -> (A -> B) -> A -> C
fact2 = fun (A B C : Type) (X : A -> B -> C) (X0 : A -> B) (X1 : A) =>
let X2 := X X1 in let X3 := X0 X1 in unkeyed (X2 X3)
  : forall A B C : (A -> B -> C) -> (A -> B) -> A -> C

Fact fact3 (m n:Z) : (1 + 2 * m <> 2 * n)%Z
fact3 - fun (m n : Z) (H : (1 + 2 * M)%Z = (2 * n)%Z) =>
let H0 :=
  fact_Zmult comm 2 m (fun x : Z => (1 + x + - (2 * n))%Z = 0%Z -> False)
   (fast_Zplus_comm 1 (m * 2)
     (fun x : Z => (x + - (2 * n))%Z = 0%Z -> False)
     (fast _Zmult_comm 2 n (fun x :Z => (M * 2 + 1 - x)%Z = 0%Z -> False)
       (fast_Zopp_mult disr r n 2
         (fun x : Z => (m * 2 + 1 + x)%Z = 0%Z -> False)
 (fast_Zplus_comm (m * 2 + 1) (n * -2)
   (fun x : Z => x = 0%Z -> False)
   (fun Omega0 : (n * -2 + (m * 2 + 1))%Z = 0%Z =>
     let H0 := erefl Gt in
     (let H1 := erefl Gt in
      fun auxiliary_1 : (1 > 0)%Z =>
      OMEGA4 1 2 (n * -1 + (m * 1 + 0)) auxiliary_1 H1
        (fast_OMEGA11 n (-1) (m * 1 + 0) 1 2
  (eq^- 0%Z)
    (fast_OMEGA11 m 1 0 1 2
```

```
      (fun x : Z => (n * (-1 * 2) + x)%Z = 0%Z)

Fact fact4: 2 + 2 = 4
fact4 =
 eq_ind_r (eq^ - 4)
  (eq_ind_r (fun _pattern_value : nat => _pattern_value_.+1 = 4)
      (eq_ind_r (fun _pattern_value : nat => _pattern_value_.+2 = 4) (erefl 4) (addn0 2))
      (addnS 2 0))
  (addnS 2 1) : 2 + 2 = 4

 \forall x,x+0 = x   \forall x y, x+S(y)=S(x+y)

2+2 = 4 is proved by

 |- \forall x,x+S(y)=S(x+y)         |- \forall x,x+S(y)=S(x+y)
 ---------------------        ---------------------
   |- 2+2 = S(2+1)                       |- S(2+1) = S(S(2+0))          |- \forall x,x+0 = x
   -----------------------------------------         ----------------
                             |- 2+2 = S(2+0)                                        |- S(2+0) = 4
      -------------------------------------------
              |- 2+2 = 4

Calculus of Construction (Coquand, Huet, 1985)
  \beta convertibility
Calculus of Inductive Construction (Coquand, Paulin, 1990)
  \beta convertibility + recursors for inductive types
Extensional Calculus of Construction (Oury 05)
  \beta + convertible
  "any terms which can be proven equal are convertible"
   (BUT proof checking isn't decidable because convertibility is not decidable)
Coq Modulo Theory (TCS'08, CSL'10, LICS'11) (Strub, Pierre Yves)
  add decision procedure to conversion rule
Calculus of Presburger Constructions
  \beta conversions (function evaluation)
  recursor for natural numbers (def. by induction of functions and types)
  a decision procedure for Presburger arithmetic in its conversion
  an extraction from the proof environment of arithmtic equations
  whose proof checking is decidable

    \Gamma |- t:T   \Gamma |- T':s'   T ~r T'
    --------------------------------
           \Gamma |- t:T'

   where ~r will include
     \beta-convertibility (function evaluation)
     recursor for natural numbers (fixpoint+match evaluation)
     validity entailment of the Presburger arithmetic
         At least, the conversion must consider equal
 any pair of Presburger-equal algebraic terms
   Algebraic terms are all terms built from 0, S,+ and variables with the right arity
   This solves our motivation examples
       list n + m ~ list m + n
   because n + m and m + n are algebraic and P-equal
   BUT this does not define a sound logic
```

B.1.19 Sutor [Suto87]

Type inference and coercion in Scratchpad II.

B.1.20 Wijngaarden [Wijn68, Section 6, p95]

ALGOL 68 has carefully defined rules for coercion, using dereferencing, deproceduring, widening, rowing, uniting, and voiding to transform values to the type required for further computation.

B.1.21 McAllester, D. and Arkondas, K., [Mcal96]

"Primitive recursion is a well known syntactic restriction on recursion definitions which guarantees termination. Unfortunately many natural definitions, such as the most common definition of Euclid's GCD algorithm, are not primitive recursive. Walther has recently given a proof system for verifying termination of a broader class of definitions. Although Walther's system is highly automatible, the class of acceptable definitions remains only semi-decidable. Here we simplify Walther's calculus and give a syntactic criteria generalizes primitive recursion and handles most of the examples given by Walthar. We call the corresponding class of acceptable defintions "Walther recursive".",

Appendix A

Untyped Lambda in Common Lisp

See Garret [Garr14]

This is a compiler for the untyped lambda calculus.

— **lambda** —

```
(defmacro ulambda (args body)
  (if (and args (atom args)) (setf args (list args)))
  (if (and (consp body) (consp (car body))) (push 'funcall body))
  (if (null args)
    body
    `(lambda (&rest args1)
  (let ((,(first args) (first args1)))
    (declare (ignorable ,(first args)))
    (flet ((,(first args) (&rest args2) (apply ,(first args) args2)))
      (if (rest args1)
        (apply (ulambda ,(rest args) ,body) (rest args1))
        (ulambda ,(rest args) ,body)))))))
```

———-

The factorial function.

— **factorial** —

```
(ulambda (n f)
  (n (ulambda (c i)
       (i (c (ulambda (f x)
               (i f (f x))))))
     (ulambda x f)
     (ulambda x x)))
```

———-

The factorial of 10. factorial of 3, add 4, compute factorial of that.

— (—

```
factorial10}
(ulambda ()
  ((ulambda (f s) (f (s (s (s (s (f (s (s (s (ulambda (f x) x)))))))))))
   (ulambda (n f) (n (ulambda (c i) (i (c (ulambda (f x) (i f (f x))))))
                   (ulambda x f) (ulambda x x)))
   (ulambda (n f x) ((n f) (f x)))
   '1+ 0))
```

Bibliography

[Acze13] Peter et al. Aczel. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.

Link: https://hott.github.io/book/nightly/hott-letter-1075-g3c53219.pdf

[Adam01] Andrew A. Adams, Martin Dunstan, Hanne Gottlieben, Tom Kelsey, Ursula Martin, and Sam Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In *Theorem proving in higher order logics*, TPHOLs 2001, pages 27–42, 2001.

Abstract: We describe an interface between version 6 of the Maple computer algebra system with the PVS automated theorem prover. The interface is designed to allow Maple users access to the robust and checkable proof environment of PVS. We also extend this environment by the provision of a library of proof strategies for use in real analysis. We demonstrate examples using the interface and the real analysis library. These examples provide proofs which are both illustrative and applicable to genuine symbolic computation problems.

[Avig14] Jeremy Avigad. LEAN proof of GCD, 2014.

Link: http://github.com/leanprover/lean2/blob/master/library/data/nat/gcd.lean

[Avig16] Jeremy Avigad. LEAN github repository, 2016.

Link: http://github.com/leanprover

[Back81] R.J.R Back. On Correct Refinement of Programs. *J. Computer and System Sciences*, 23(1):49–68, 1981.

Abstract: The stepwise refinement technique is studied from a mathematical point of view. A relation of correct refinement between programs is defined, based on the principle that refinement steps should be correctness preserving. Refinement between programs will therefore depend on the criterion of program correctness used. The application of the refinement relation in showing the soundness of different techniques for refining programs is discussed. Special attention is given to the use of abstraction in program construction. Refinement with respect to partial and total correctness will be studied in more detail, both for deterministic and nondeterministic programs. The relationship between these refinement relations and the approximation relation of fixpoint semantics will be studied, as well as the connection with the predicate transformers used in program verification.

[Ball95] Clemens Ballarin, Karsten Homann, and Jacques Calmet. Theorems and Algorithms: An Interface between Isabelle and Maple. In *ISSAC 95*, pages 150–157. ACM, 1995.

Abstract: Solving sophisticated mathematical problems often requires algebraic algorithms *and* theorems. However, there are no environments integrating theorem provers and computer algebra systems which consistently provide the inference capabilities of the first and the powerful arithmetic of the latter systems. As an example for such a mechanized mathematics environment we describe a prototype implementation of an interface between Isabelle and Maple. It is achieved by extending the simplifier of Isabelle through the introduction of a new class of simplification rules called evaluation rules in order to make selected operations of Maple available, and without any modification to the computer algebra system. Additionally, we specify syntax translations for the concrete syntax of Maple which enables the communication between both systems illustrated by some examples that can be solved by theorems and algorithms

Link: https://pdfs.semanticscholar.org/077e/606f92b4095637e624a9efc942c5c63c4bc2.pdf

[Bate85] Joseph L. Bates and Robert L. Constable. Proofs as Programs. *ACM TOPLAS*, 7(1), 1985.

Abstract: The significant intellectual cost of programming is for problem solving and explaining, not for coding. Yet programming systems offer mechanical assistance for the coding process exclusively. We illustrate the use of an implemented program development system, called PRL ('pearl'), that provides automated assistance with the difficult part. The problem and its explained solution are seen as formal objects in a constructive logic of the data domains. These formal explanations can be executed at various stages of completion. The most incomplete explanations resemble applicative programs, the most complete are formal proofs.

[Berg95] U. Berger and H. Schwichtenberg. The Greatest Common Divisor: A Case Study for Program Extraction from Classical Proofs. *LNCS*, 1158:36–46, 1995.

[Blak96] Bob Blakley. The Emperor's Old Armor. In *Proc. 1996 New Security Paradigms Workshop*. ACM, 1996.

[Bold07] Sylvie Boldo and Jean-Christophe Filliatre. Formal Verification of Floating-Point programs.

Link: http://www-lipn.univ-paris13.fr/CerPAN/files/ARITH.pdf

[Bold07a] Sylvie Boldo and Jean-Christophe Filliatre. Formal Verification of Floating-Point programs.

Abstract: This paper introduces a methodology to perform formal verification of floating-point C programs. It extends an existing tool for verification of C programs, Caduceus, with new annotations for specific floating-point arithmetic. The Caduceus first-order logic model for C programs is extended accordingly. Then verification conditions are obtained in the usual way and can be discharged interactively with the Coqa proof assistant, using an existing Coq formalization of floating-point arithmetic. This methodology is already implemented and has been successfully applied to several short floating-point programs, which are presented in this paper.

Link: http://www.lri.fr/~filliatr/ftp/publis/caduceus-floats.pdf

[Bold11] Sylvie Boldo and Claude Marche. Formal verification of numerical programs: from C annotated programs to mechanical proofs. *Mathematics in Computer Science*, 5:377–393, 2011.

Abstract: Numerical programs may require a high level of guarantee. This can be achieved by applying formal methods, such as machine-checked proofs. But these tools handle mathematical theorems while we are interested in C code, in which numerical computations are performed using floating-point arithmetic, whereas proof tools typically handle exact real arithmetic. To achieve this high level of confidence on C programs, we use a chain of tools: Frama-C, its Jessie plugin, Why and provers among Coq, Gappa, Alt-Ergo, CVC3 and Z3. This approach requires the C program to be annotated; each function must be precisely specified, and we prove the correctness of the program by proving both that it meets its specifications and that no runtime error may occur. The purpose of this paper is to illustrate, on various examples, the features of this approach.

Link: https://hal.archives-ouvertes.fr/hal-00777605/document

[Book102] Axiom Authors. *Volume 10.2: Axiom Algebra: Categories.* Axiom Project, 2016.

Link: http://axiom-developer.org/axiom-website/bookvol10.2.pdf

[Book103] Axiom Authors. *Volume 10.3: Axiom Algebra: Domains.* Axiom Project, 2016.

Link: http://axiom-developer.org/axiom-website/bookvol10.3.pdf

[Bowe95] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12(4):34–41, 1995.

Abstract: New myths about formal methods are gaining tacit acceptance both outside and inside the system-development community. The authors address and dispel these myths based on their observations of industrial projects. The myths include: formal methods delay the development process; they lack tools; they replace traditional engineering design methods; they only apply to software; are unnecessary; not supported; and formal methods people always use formal methods.

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

[Broo87] Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, 1987.

Abstract: Fashioning complex conceptual constructs is the essence; accidental tasks arise in representing the constructs in language. Past progress has so reduced the accidental tasks that future progress now depends upon addressing the essence.

[Buch97] Bruno Buchberger. Mathematica: doing mathematics by computer? *Advances in the design of symbolic computation systems*, pages 2–20, 1997, 978-3-211-82844-1.

[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.

[COQnat] COQ Proof Assistant. Library Coq.Init.Nat, 2017.

Abstract: Peano natural numbers, defintions of operations

Link: https://coq.inria.fr/library/Coq.Init.Nat.html

[Cald97] James L. Caldwell. Moving proofs-as-programs into practice. In *Automated Software Engineering*. IEEE, 1997.

Abstract: Proofs in the Nuprl system, an implementation of a constructive type theory, yield correct-by-construction programs. In this paper a new methodology is presented for extracting efficient and readable programs from inductive proofs. The resulting extracted programs are in a form suitable for use in hierarchical verifications in that they are amenable to clean partial evaluation via extensions to the Nuprl rewrite system. The method is based on two elements: specifications written with careful use of the Nuprl set-type to restrict the extracts to strictly computational content; and on proofs that use induction tactics that generate extracts using familiar fixed-point combinators of the untyped lambda calculus. In this paper the methodology is described and its application is illustrated by example.

[Calu07] C.S. Calude, E. Calude, and S. Marcus. Proving and Programming. technical report CDMTCS-309, Centre for Discrete Mathematics and Theoretical Computer Science, 2007.

Abstract: There is a strong analogy between proving theorems in mathematics and writing programs in computer science. This paper is devoted to an analysis, from the perspective of this analogy, of proof in mathematics. We will argue that while the Hilbertian notion of proof has few chances to change, future proofs will be of various types, will play different roles, and their truth will be checked differently. Programming gives mathematics a new form of understanding. The computer is the driving force behind these changes.

[Card85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985.

Abstract: Our objective is to understand the notion of type in programming languages, present a model of typed, polymorphic programming languages that reflects recent research in type theory, and examine the relevance of recent research to the design of practical programming languages. Object-oriented languages provide both a framework and a motivation for exploring the interaction among the concepts of type, data abstraction, and polymorphism, since they extend the notion of type to data abstraction and since type inheritance is an important form of polymorphism. We develop a λ-calculus-based model for type systems that allows us to explore these interactions in a simple setting, unencumbered by complexities of production programming languages. The evolution of languages from untyped universes to monomorphic and then polymorphic type systems is reviewed. Mechanisms for polymorphism such as overloading, coercion, subtyping, and parameterization are examined. A unifying framework for polymorphic type systems is developed in terms of the typed λ-calculus augmented to include binding of types by quantification as well as binding of values by abstraction. The typed λ-calculus is augmented by universal quantification to model generic functions with type parameters, existential quantification and packaging (information hiding) to model abstract data types, and bounded quantification to model subtypes and type inheritance. In this way we obtain a simple and precise characterization of a powerful type system that includes abstract data types, parametric polymorphism, and multiple inheritance in a single consistent framework. The mechanisms for type checking for the augmented λ-calculus are discussed. The augmented typed λ-calculus is used as a programming language for a variety of illustrative examples. We christen this language Fun because fun instead of λ is the functional abstraction keyword and because it is pleasant to deal with. Fun is mathematically simple and can serve as a basis for the design and implementation of real programming languages with type facilities that are more powerful and expressive than those of existing programming languages.

In particular, it provides a basis for the design of strongly typed object-oriented languages

[Cast16] Pierre Casteran and Mattieu Sozeau. A Gentle Introduction to Type Classses and Relations in Coq, 2016.

Link: http://www.labri.fr/perso/casteran/CoqArt/TypeClassesTut/typeclassestut.pdf

[Chli15] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2015, 9780262026659.

Link: http://adam.chlipala.net/cpdt/cpdt.pdf

[Chli17] Adam Chlipala. *Formal Reasoning About Programs*. MIT, 2017.

Abstract: Briefly, this book is about an approach to bringing software engineering up to speed with more traditional engineering disciplines, providing a mathematical foundation for rigorous analysis of realistic computer systems. As civil engineers apply their mathematical canon to reach high certainty that bridges will not fall down, the software engineer should apply a different canon to argue that programs behave properly. As other engineering disciplines have their computer-aided design tools, computer science has proof assistants, IDEs for logical arguments. We will learn how to apply these tools to certify that programs behave as expected. More specifically: Introductions to two intertangled subjects: the Coq proof assistant, a tool for machine-checked mathematical theorem proving; and formal logical reasoning about the correctness of programs.

Link: http://adam.chlipala.net/frap/frap_book.pdf

[Chli17a] Adam Chlipala. Coming Soon: Machine-Checked Mathematical Proofs in Everyday Software and Hardware Development, 2017.

Abstract: Most working engineers view machine-checked mathematical proofs as an academic curiosity, if they have ever heard of the concept at all. In contrast, activities like testing, debugging, and code review are accepted as essential. They are woven into the lives of nearly all developers. In this talk, I will explain how I see machine-checked proofs enabling new everyday activities for developers of computer software and hardware. These activities have the potential to lower development effort dramatically, at the same time as they increase our assurance that systems behave correctly and securely. I will give a cosmological overview of this field, answering the FAQs that seem to stand in the way of practicality; and I will illustrate the principles with examples from projects that you can clone from GitHub today, covering the computing stack from digital hardware design to cryptographic software and applications. Today's developers of computer software and hardware are tremendously effective, compared to their predecessors. We have found very effective ways of modularizing and validating our work. The talk is about ammunition for these activities from a perhaps-unexpected source.

Modularity involves breaking a complex system into a hierarchy of simpler pieces, which may be written and understood separately. Structured programming (e.g., using loops and conditionals instead of gotos) helps us read and understand parts of a single function in isolation, and data abstraction lets us encapsulate important functionality in objects, with guarantees that other code can only access the private data by calling public methods. That way, we can convince ourselves that the encapsulated code upholds certain essential properties, regardless of which other code it is linked with. Systematic unit testing also helps enforce contracts for units of modularity. Each of these techniques can be rerun automatically, to catch regressions in evolving systems, and catch those regressions in a way that accurately points the finger of responsibility to particular modules. Validation is an important part of development that encompasses testing, debugging, code review, and anything else that we do to raise our confidence that the system behaves as intended. Experienced engineers know that validation tends to take up the majority of engineering effort. Often that effort involves mentally taxing activities that would not otherwise come up in coding. One example is thinking about test-case coverage, and another is including instrumentation that produces traces to consult during debugging. It is not hard for working developers to imagine great productivity gains from better ways to break systems into pieces or raise our confidence in those pieces. The claim I will make in this talk is that a key source of such insights has been neglected: machine-checked mathematical proofs. Here the basic functionality is an ASCII language for defining mathematical objects, stating theorems about them, and giving proofs of theorems. Crucially, an algorithm checks that purported proofs really do establish the theorems. By going about these activities in the style of programming, we inherit usual supporting tools like IDEs, version control, continuous integration, and automated build processes. But how could so esoteric a task as math proofs call for that kind of tooling, and what does it have to do with building real computer systems? I will explain a shared vision to that end, developed along with many other members of my research community. Let me try to convince you that all of the following goals are attainable in the next 10 years.

- We will have complete computer systems implementing moderately complex network servers for popular protocols, proved to implement those protocols correctly, from the level of digital circuits on up. We will remove all deployed code (hardware or software) from the trusted computing base, shifting our trust to much smaller specifications and proof checkers.

- Hobbyists will be able to design new embedded computing platforms by mixing and matching open-source hardware and software components, also mixing and matching

the proofs of these components, guaranteeing no bugs at the digital-abstraction level or higher, with no need for debugging.

- New styles of library design will be enabled by the chance to attach a formal behavioral specification to each library. For instance, rank-and-file programmers will able to assemble their own code for cryptographic protocols, with code that looks like reference implementations in Python, but getting performance comparable to what experts handcraft in assembly today. Yet that benefit would come with no need to trust that library authors have avoided bugs or intentional backdoors, perhaps even including automatic proofs of cryptographic security properties.

Main technical topics to cover to explain my optimism:

- The basic functionality of proof assistants and why we should trust their conclusions
- How to think about system decomposition with specifications and proofs, including why, for most components, we do not need to worry about specification mistakes
- The different modes of applying proof technology to check or generate components
- The engineering techniques behind cost-effective proof authoring for realistic systems
- A hardware case study: Kami, supporting component-based digital hardware authoring with proofs
- A software case study: Fiat Cryptography, supporting correct-by-construction auto-generation of fast code for elliptic-curve cryptography
- Pointers to where to look next, if you would like to learn more about this technology

Link: https://media.ccc.de/v/34c3-9105-coming_soon_machine-checked_mathematical_proofs_in_everyday_software_and_hardware_development

[Clar91] Edmund Clarke and Xudong Zhao. Analytica – A Theorem Prover in Mathematica, 1991.

Link: http://www.cs.cmu.edu/~emc/papers/Conference%20Papers/Analytica%20A%20Theorem%20Prover%20in%20Mathematica.pdf

[Cons98] Robert L. Constable and Paul B. Jackson. Towards Integrated Systems for Symbolic Algebra and Formal Constructive Mathematics, 1998.

Abstract: The purpose of this paper is to report on our efforts to give a formal account of some of the algebra used in Computer Algebra Systems (CAS). In particular, we look at the concepts used in the so called 3rd generation algebra systems, such as

Axiom[4] and Weyl[9]. It is our claim that the Nuprl proof development system is especially well suited to support this kind of mathematics.

Link: http://www.nuprl.org/documents/Constable/towardsintegrated.pdf

[Coqu16] Thierry Coquand, Gérard Huet, and Christine Paulin. The COQ Proof Assistant, 2016.

Link: https://coq.inria.fr

[Coqu16a] Thierry Coquand, Gérard Huet, and Christine Paulin. COQ Proof Assistant Library Coq.ZArith.Znumtheory, 2016.

Link: https://coq.inria.fr/library/Coq.ZArith.Znumtheory.html

[Coqu86] Thierry Coquand and Gérard Huet. The Calculus of Constructions. Technical Report 530, INRIA Centre de Rocquencourt, 1986.

Abstract: The Calculus of Constructions is a higher-order formalism for constructive proofs in natural deduction style. Every proof is a λ-expression, typed with propositions of the underlying logic. By removing types we get a pure λ-expression, expressing its associated algorithm. Computing this λ-expression corresponds roughly to cut-elimination. It is our thesis that (as already advocated by Martin-Lof) the Curry-Howard correspondence between propositions and types is a powerful paradigm for Computer Science. In the case of Constructions, we obtain the notion of a very high-level functional programming language, with complex polymorphism well-suited for modules specification. The notion of type encompasses the usual notioin of data type, but allows as well arbitrarily complex algorithmic specifications. We develop the basic theory of a Calculus of Constructions, and prove a strong normalization theorem showing that all computations terminate. Finally, we suggest various extensions to stronger calculi.

Link: https://hal.inria.fr/inria-00076024/document

[Croc14] David Crocker. Can C++ Be Made as Safe as SPARK? In *Proc 2014 HILT*, 2014, 978-1-4503-3217-0.

Abstract: SPARK offers a way to develop formally-verified software in a language (Ada) that is designed with safety in mind and is further restricted by the SPARK language subset. However, much critical embedded software is developed in C or C++ We look at whether and how benefits similar to those offered by the SPARK language subset and associated tools can be brought to a C++ development environment.

[Cyph17] Cypherpunks. Chapter 4: Verification Techniques, 2017.

Abstract: Wherein existing methods for building secure systems are examined and found wanting

Link: http://www.cypherpunks.to/~peter/04_verif_ techniques.pdf

[Dave02] James H. Davenport. Equality in computer algebra and beyond. *J. Symbolic Computing*, 34(4):259–270, 2002.

Abstract: Equality is such a fundamental concept in mathematics that, in fact, we seldom explore it in detail, and tend to regard it as trivial. When it is shown to be non-trivial, we are often surprised. As is often the case, the computerization of mathematical computation in computer algebra systems on the one hand, and mathematical reasoning in theorem provers on the other hand, forces us to explore the issue of equality in greater detail.In practice, there are also several ambiguities in the definition of equality. For example, we refer to $\mathbb{Q}(x)$ as "rational functions", even though $\frac{x^2-1}{x-1}$ and $x+1$ are not equal as functions from $\mathbb{R}$ to $\mathbb{R}$, since the former is not defined at $x = 1$, even though they are equal as elements of $\mathbb{Q}(x)$. The aim of this paper is to point out some of the problems, both with mathematical equality and with data structure equality, and to explain how necessary it is to keep a clear distintion between the two.

Link: http://www.calculemus.net/meetings/siena01/Papers/Davenport.pdf

[Davi09] Jared Curran Davis. *A Self-Verifying Theorem Prover*. PhD thesis, University of Texas at Austin, 2009.

Abstract: Programs have precise semantics, so we can use mathematical proof to establish their properties. These proofs are often too large to validate with the usual social process of mathematics, so instead we create and check them with theorem-proving software. This software must be advanced enough to make the proof process tractable, but this very sophistication casts doubt upon the whole enterprise: who verifies the verifier? We begin with a simple proof checker, Level 1, that only accepts proofs composed of the most primitive steps, like Instantiation and Cut. This program is so straightforward the ordinary, social process can establish its soundness and the consistency of the logical theory it implements (so we know theorems are always true). Next, we develop a series of increasingly capable proof checkers, Level 2, Level 3, etc. Each new proof checker accepts new kinds of proof steps which were not accepted in the previous levels. By taking advantage of these new proof steps, higher-level proofs can be written more concisely than lower-level proofs, and can take less time to construct and check. Our highest-level proof checker, Level 11, can be thought of as a simplified version of the ACL2 or NQTHM theorem provers. One contribution of this work is to show how such systems can be verified. To establish that the Level 11 proof checker can be trusted, we first use it, without trusting it, to prove the fidelity of every Level n to Level 1: whenever Level n accepts a proof of some ϕ, there exists a Level 1 proof of ϕ. We then mechanically translate the Level 11 proof for each Level n into a Level n 1 proofthat is, we create a Level 1 proof of Level 2s fidelity, a Level 2 proof of Level 3s fidelity, and so on. This layering shows that each level can be trusted, and allows us to manage the sizes of these proofs. In this way, our system proves its own fidelity, and trusting Level 11 only requires us to trust Level 1.

[Davi15] Jared Davis and Magnus O. Myreen. The Reflective Milawa Theorem Prover is Sound. *J. Automated Reasoning*, 55(2):117–183, 2015.

Abstract: This paper presents, we believe, the most comprehensive evidence of a theorem provers soundness to date. Our subject is the Milawa theorem prover. We present evidence of its soundness down to the machine code. Milawa is a theorem prover styled after NQTHM and ACL2. It is based on an idealised version of ACL2s computational logic and provides the user with high-level tactics similar to ACL2s. In contrast to NQTHM and ACL2, Milawa has a small kernel that is somewhat like an LCF-style system. We explain how the Milawa theorem prover is constructed as a sequence of reflective extensions from its kernel. The kernel establishes the soundness of these extensions during Milawas boot-strapping process. Going deeper, we explain how we have shown that the Milawa kernel is sound using the HOL4 theorem prover. In HOL4, we have formalized its logic, proved the logic sound, and proved that the source code for the Milawa kernel (1,700 lines of Lisp) faithfully implements this logic. Going even further, we have combined these results with the x86 machine-code level verification of the Lisp runtime Jitawa. Our top-level theorem states that Milawa can never claim to prove anything that is false when it is run on this Lisp runtime.

[Demi79] Richard A. DeMilo, Richard J. Lipton, and Alan J. Perlis. Social Processes and Proofs of Theorems and Programs. *Communications of the ACM*, 22(5):271–280, 1979.

Abstract: It is argued that formal verifications of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics. Furthermore the absence of continuity, the inevitability of change, and the complexity of specification of significantly many real programs make the formal verification process difficult to justify and manage. It is felt that ease of formal verification should not dominate program language design.

[Dijk83] Edsger Dijkstra. Fruits of Misunderstanding, 1983.

Link: `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD854.html`

[Domi18] Domipheus. Designing a CPU in VHDL, 2018.

Abstract: A VHDL CPU

Link: `http://labs.domipheus.com/blog/tpu-series-quick-links`

[Fent93] Norman Fenton. How Effective Are Software Engineering Methods. *J.*

Systems Software, 22:141–148, 1993.

Abstract: For 25 years, software engineers have sought methods which they hope can provide a technological fix for the software crisis. Proponents of specific methods claim that their use leads to significantly improved quality and productivity. Such claims are rarely, if ever, backed up by hard evidence. We show that often, where real empirical evidence does exist, the results are counter to the views of the so-called experts. We examine the impact on the software industry of continuing to place our trust in unproven, and often revolutionary, methods. The very poor state of the art of empirical assessment in software engineering can be explained in part by inappropriate or inadequate use of measurement. Numerous empirical studies are flawed because of their poor experimental design and lack of adherence to proper measurement principles.

[Fetz88] James H. Fetzer. Program Verification: The Very Idea. *Communications of the ACM*, 31(9):1048–1063, 1988.

Abstract: The notion of program verification appears to trade upon an equvocation. Algorithms, as logical structures, are appropriate subjects for deductive verification. Programs, as causal models of those structures, are not. The success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility.

[Fill03] Jean-Christophe Filliatre. Verification of Non-Functional Programs using Interpretations in Type Theory. *J. Functional Programming*, 13(4):709–745, 2003.

Abstract: We study the problem of certifying programs combining imperative and functional features within the general framework of type theory. Type theory is a powerful specification language, which is naturally suited for the proof of purely functional programs. To deal with imperative programs, we propose a logical interpretation of an annotated program as a partial proof of its specification. The construction of the corresponding partial proof term is based on a static analysis of the effects of the program which excludes aliases. The missing subterms in the partial proof term are seen as proof obligations, whose actual proofs are left to the user. We show that the validity of those proof obligations implies the total correctness of the program. This work has been implemented in the Coq proof assistant. It appears as a tactic taking an annotated program as argument and generating a set of proof obligations. Several nontrivial algorithms have been certified using this tactic.

[Fill13] Jean-Christophe Filliatre and Andrei Paskevich. Why3 – Where Programs Meet Provers. *LNCS*, 7792, 2013.

Abstract: We present Why3, a tool for deductive program verification, and WhyML, its programming and specification language. WhyML is a first-order language with polymorphic types, pattern matching, and inductive predicates. Programs can make use of record types with mutable fields, type invariants, and ghost code. Verification conditions are discharged by Why3 with the help of various existing automated and interactive theorem provers. To keep verification conditions tractable and comprehensible, WhyML imposes a static control of aliases that obviates the use of a memory model. A user can write WhyML programs directly and get correct-by-construction OCaml programs via an automated extraction mechanism. WhyML is also used as an intermediate language for the verification of C, Java, or Ada programs. We demonstrate the benefits of Why3 and WhyML on non-trivial examples of program verification.

Link: `https://hal.inria.fr/hal-00789533/document`

[Floy67] Robert W. Floyd. Assigning Meanings to Programs. In *Proc. Symp. in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[Frad08] Maria Joao Frade. Calculus of Inductive Construction. Software Formal Verification. *MFES*, 2008.

Link: `http://www4.di.uminho.pt/~jno/mfes/0809/SFV-CIC.pdf`

[Freg1891] Gottlob Frege. Function and Concept, 1891.

Link: `http://fitelson.org/proseminar/frege_fac.pdf`

[Frue91] Thom Fruehwirth, Ehud Shapiro, Moshe Y. Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Proc. Sixth Annual IEEE Symp. on Logic in Comp. Sci.*, pages 300–309. IEEE, 1991.

Abstract: Type checking can be extremely useful to the program development process. Of particular interest are descriptive type systems, which let the programmer write programs without having to define or mention types. We consider here optimistic type systems for logic programs. In such systems types are conservative approximations to the success set of the program predicates. We propose the use of logic programs to describe types. We argue that this approach unifies the denotational and operational approaches to descriptive type systems and is simpler and more natural than previous approaches. We focus on the use of unary-predicate programs to describe types. We identify a proper class of unary-predicate programs and show that it is expressive enough to express several notions of types. We use an analogy with 2-way automata and a correspondence with alternating algorithms to obtain a complexity characterization of type inference and type checking. This characterization was facilitated by the use of logic programs to represent types.

[Garr14] Ron Garret. The Awesome Power of Theory, 2014.

Link: `http://www.flownet.com/ron/lambda-calculus.html`

[Glas02] Robert L. Glass. The Proof of Correctness Wars. *Communications of the ACM*, 45(8):19–21, 2002.

[Hall90] Anthony Hall. 7 Myths of Formal Methods. *IEEE Software*, 7(5):11–19, 1990.

Abstract: Formal methods are difficult, expensive, and not widely useful, detractors say. Using a case study and other real-world examples, this article challenges such common myths.

[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.

Link: http://www.jrmcclurg.com/papers/law_2013_paper.pdf

[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.

Link: http://arxiv.org/pdf/1406.1566

[Harp13] Robert Harper. 15.819 Homotopy Type Theory Course, 2013.

Link: http://www.cs.cmu.edu/~rwh/courses/hott

[Harr98] J. Harrison and L. Thery. A Skeptic's approach to combining HOL and Maple. *J. Autom. Reasoning*, 21(3):279–294, 1998.

Abstract: We contrast theorem provers and computer algebra systems, pointing out the advantages and disadvantages of each, and suggest a simple way to achieve a synthesis of some of the best features of both. Our method is based on the systematic separation of search for a solution and checking the solution, using a physical connection between systems. We describe the separation of proof search and checking in some detail, relating it to proof planning and to the complexity class NP, and discuss different ways of exploiting a physical link between systems. Finally, the method is illustrated by some concrete examples of computer algebra results proved formally in the HOL theorem prover with the aid of Maple.

Link: http://www.cl.cam.ac.uk/~jrh13/papers/cas.ps.gz

[Hoar69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *CACM*, 12(10):576–580, 1969.

Abstract: In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics

Link: https://www.cs.cmu.edu/~crary/819-f09/Hoare69.pdf

[Hoar87] Charles Antony Richard Hoare. An Overview of Some Formal Methods for Program Design. *Computer*, 20(9), 1987.

[Hoar96] C.A.R Hoare. How did software get so reliable without proof? *LNCS*, 1051, 1996.

Abstract: By surveying current software engineering practice, this paper reveals that the techniques employed to achieve reliability are little different from those which have proved effective in all other branches of modern engineering: rigorous management of procedures for design inspection and review; quality assurance based on a wide range of targeted tests; continuous evolution by removal of errors from products already in widespread use; and defensive programming, among other forms of deliberate overengineering. Formal methods and proof play a small direct role in large scale programming; but they do provide a conceptual framework and basic understanding to promote the best of current practice, and point directions for future improvement.

[Homa94] Karsten Homann and Jacques Calmet. Combining Theorem Proving and Symbolic Mathematical Computing. *LNCS*, 958:18–29, 1994.

Abstract: An intelligent mathematical environment must enable symbolic mathematical computation and sophisticated reasoning techniques on the underlying mathematical laws. This paper disscusses different possible levels of interaction between a symbolic calculator based on algebraic algorithms and a theorem prover. A high level of interaction requires a common knowledge representation of the mathematical knowledge of the two systems. We describe a model for such a knowledge base mainly consisting of type and algorithm schemata, algebraic algorithms and theorems.

[Jack95] Paul Bernard Jackson. *Enhancing the NUPRL Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Cornell University, 1 1995.

Abstract: This thesis describes substantial enhancements that were made to the software tools in the Nuprl system that are used to interactively guide the production of formal proofs. Over 20,000 lines of code were written for these tools. Also, a corpus of formal mathematics was created that consists of roughly 500 definitions and 1300 theorems. Much of this material is of a foundational nature and supports all current work in Nuprl. This thesis concentrates on describing the half of this corpus that is concerned with abstract algebra and that covers topics central to the mathematics of the computations carried out by computer algebra systems. The new proof tools include those that solve linear arithmetic problems, those that apply the properties of order relations, those that carry out inductive proof to support recursive definitions, and those that do sophisticated rewriting. The rewrite tools allow rewriting with relations of differing strengths and take care of selecting and applying appropriate congruence lemmas automatically. The rewrite relations can be order relations as well as equivalence relations. If they are order relations, appropriate monotonicity lemmas are selected. These proof tools were heavily used throughout the work on computational algebra. Many examples are given that illustrate their operation and demonstrate their effectiveness. The foundation for algebra introduced classes of monoids, groups, ring and modules, and included theories of order relations and permutations. Work on finite sets and multisets illustrates how a quotienting operation hides details of datatypes when reasoning about functional programs. Theories of summation operators were developed that drew indices from integer ranges, lists and multisets, and that summed over all the classes mentioned above. Elementary factorization theory was developed that characterized when cancellation monoids are factorial. An abstract data type for the operations of multivariate polynomial arithmetic was defined and the correctness of an implementation of these operations was verified. The implementation is similar to those found in current computer algebra systems. This work was all done in Nuprl's constructive type theory. The thesis discusses the appropriateness of this foundation, and the extent to which the work relied on it.

Keyword: axiomref, CAS-Proof, printed

[Jenk84b] Richard D. Jenks. A primer: 11 keys to New Scratchpad. In *Proc. EUROSAM ISSAC 1984*, pages 123–147. Springer-Verlag, 1984, 0-387-13350-X.

Abstract: This paper is an abbreviated primer for the language of new SCRATCHPAD, a new implementation of SCRATCHPAD which has been under design and development by the Computer Algebra Group at the IBM Research Center during the past 6 years. The basic design goals of the new SCRATCHPAD language and interface to the user are to provde:

- a "typeless" interactive language suitable for on-line solution of mathematical problems by novice users with little or no programming required, and
- a programming language suitable for the formal description of algorithms and algebraic structures which can be compiled into run-time efficient object code.

The new SCRATCHPAD language is introduced by 11 keys with each successive key introducing an additional capability of the language. The language is thus described as a "concentric" language with each of the 11 levels corresponding to a language subset. These levels are more than just a pedagogic device, since they correspond to levels at which the system can be effectively used. Level 1 is sufficient for naive interactive use; levels 2-8 progressively introduce interactive users to capabilities of the language; levels 9-11 are for system programmers and advanced users. Levesl 2, 4, 6, and 7 give users the full power of LISP with a high-level language; level 8 introduces "type declarations;" level 9 allows polymorphic functions to be defined and compiled; levels 10-11 give users an Ada-like facility for defining types and packages (those of new SCRATCHPAD are dynamically constructable, however). One language is used for both interactive and system programming language use, although several freedomes such as abbreviation and optional type-declarations allowed at top-level are not permitted in system code. The interactive language (levels 1-8) is a blend of original SCRATCHPAD [GRJY75], some proposed extensions [JENK74], work by Loos [LOOS74], SETL [DEWA79], SMP [COWO81], and new ideas; the system programming language (levels 1-11) superficially resembles Ada but is more similar to CLU [LISK74] in its semantic design. This presentation of the language in this paper omits many details to be covered in the SCRATCHPAD System Programming Manual [SCRA84] and an expanded version of this paper will serve as a primer for SCRATCHPAD users [JESU84].

[Juds15] Thomas W. Judson. *Abstract Algebra: Theory and Applications.* Website, 2015.

Link: `http://abstract.ups.edu/aata/colophon-1.html`

[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.

[Kife91] Michael Kifer and James Wu. A First-order Theory of Types and Polymorphism in Logic Programming. In *Proc Sixth Annual IEEE Symp. on Logic in Comp. Sci.*, pages 310–321, 1991.

Abstract: A logic called typed predicate calculus (TPC) that gives declarative meaning to logic programs with type declarations and type inference is introduced. The proper interaction between parametric and inclusion varieties of polymorphism is achieved through a construct called type dependency, which is analogous to implication types but yields more natural and succinct specifications. Unlike other proposals where typing has extra-logical status, in TPC the notion of type-correctness has precise model-theoretic meaning that is independent of any specific type-checking or type-inference procedure. Moreover, many different approaches to typing that were proposed in the past can be studied and compared within the framework of TPC. Another novel feature of TPC is its reflexivity with respect to type declarations; in TPC, these declarations can be queried the same way as any other data. Type reflexivity is useful for browsing knowledge bases and, potentially, for debugging logic programs.

[Kreb17] Robbert Jan Krebbers. The CH_2O formalization of ISO C11, 2017.

Link: http://robbertkrebbers.nl/research/ch2o/

[Lamp02] Leslie Lamport. *Specifying Systems*. Addison-Wesley, 2002, 0-321-14306-X.

Link: http://research.microsoft.com/en-us/um/people/lamport/tla/book-02-08-08.pdf

[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.

Link: http://research.microsoft.com/en-us/um/people/lamport/pubs/paper.pdf

[Lamp14a] Leslie Lamport. Talk: How to Write a 21^{st} Century Proof, 2014.

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

Link: http://hits.mediasite.com/mediasite/Play/29d825439b3c49f088d35555426fbdf81d

[Lamp16] Leslie Lamport. TLA+ Proof System, 2016.

Abstract: Demonstration of Euclid Algorithm Proof in TLA+

Link: https://tla.msr-inria.inria.fr/tlaps/content/Documentation/Tutorial/The_example.html

[Mahb16] Assia Mahboubi, Enrico Tassi, Yves Bertot, and Georges Gonthier. *Mathematical Components*. math-comp.github.io/mcb, 2016.

Abstract: *Mathematical Components* is the name of a library of formalized mathematic for the COQ system. It covers a veriety of topics, from the theory of basic data structures (e.g. numbers, lists, finite sets) to advanced results in various flavors of algebra. This library constitutes the infrastructure for the machine-checked proofs of the Four Color Theorem and the Odd Order Theorem. The reason of existence of this books is to break down the barriers to entry. While there are several books around covering the usage of the COQ system and the theory it is based on, the Mathematical Components library is build in an unconventional way. As a consequence, this book provides a non-standard presentation of COQ, putting upfront the formalization choices and the proof style that are the pillars of the library. This book targets two classes of public. On one hand, newcomers, even the more mathematically inclined ones, find a soft introduction to the programming language of COQ, Gallina, and the Ssreflect proof language. On the other hand accustomed COQ users find a substantial accound of the formalization style that made the Mathematical Components library possible. By no means does this book pretend to be a complete description of COQ or Ssreflect: both tools already come with a comprehensive user manual. In the course of the book, the reader is nevertheless invited to experiment with a large library of formalized concepts and she is given as soon as possible sufficient tools to prove non-trivial mathematical results by reusing parts of the library. By the end of the first part, the reader has learnt how to prove formally the infinitude of prime numbers, or the correctnes of the Euclidean's division algorithm, in a few lines of proof text.

Link: https://math-comp.github.io/mcb/book.pdf

[Mann78] Zohar Manna and Richard Waldinger. The Logic of Computer Programming. *IEEE Trans. on Software Engineering*, 4(3), 1978.

Abstract: Techniques derived from mathematical logic promise to provide an alternative to the conventional methodology for constructing, debugging, and optimizing computer programs. Ultimately, these techniques are intended to lead to the automation of many of the facets of the programming process. This paper provides a unified tutorial exposition of the logical techniques, illustrating each with examples. The strengths and limitations of each technique as a practial programming aid are assessed and attempts to implement these methods in experimental systems are discussed.

[Mart79] P. Martin-Lof. Constructive Mathematics and Computer Programming. In *Proc. 6th Int. Congress for Logic, Methodology and Philosophy of Science*, pages 153–179. North-Holland, 1979.

Abstract: If programming is understood not as the writing of instructions for this or that computing machine but as the design of methods of computation that it is the computer's duty to execute (a difference that Dijkstra has referred to as the difference between computer science and computing science), then it no longer seems possible to distinguish the discipline of programming from constructive mathematics. This explains why the intuitionistic theory of types, which was originally developed as a symbolism for the precise codification of constructive mathematics, may equally well be viewed as a programming language. As such it provides a precise notation not only, like other programming languages, for the programs themselves but also for the tasks that the programs are supposed to perform. Moreover, the inference rules of the theory of types, which are again completely formal, appear as rules of correct program synthesis. Thus the correctness of a program written in the theory of types is proved formally at the same time as it is being synthesized.

[Maso86] Ian A. Mason. *The Semantics of Destructive Lisp.* Center for the Study of Language and Information, 1986, 0-937073-06-7.

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.

[Mcal96] D. McAllester and K. Arkondas. Walther Recursion. In *CADE 13*. Springer-Verlag, 1996.

Abstract: Primitive recursion is a well known syntactic restriction on recursion definitions which guarantees termination. Unfortunately many natural definitions, such as the most common definition of Euclid's GCD algorithm, are not primitive recursive. Walther has recently given a proof system for verifying termination of a broader class of definitions. Although Walther's system is highly automatible, the class of acceptable definitions remains only semi-decidable. Here we simplify Walther's calculus and give a syntactic criteria generalizes primitive recursion and handles most of the examples given by Walthar. We call the corresponding class of acceptable defintions "Walther recursive".

[Mein13] Diana Meindl. Implementation of an Algorithm Computing the Greatest Common Divisor for Multivariate Polynomials. Master's thesis, RISC Linz, 2013.

[Mesh16a] Sergei D. Meshveliani. Provable programming of algebra: particular points, examples, 2016.

Abstract: It is discussed an experiance in provable programming of a computer algebra library with using a purely functional language with dependent tyhpes (`Agda`). There are given several examples illustrating particular points of implementing the approach of constructive mathematics.

Link: http://www.botik.ru/pub/local/Mechveliani/provProgExam.zip

[Myre09] Magnus O. Myreen and Michael J.C. Gordon. Verified LISP Implementations on ARM, x86 and PowerPC. *LNCS*, 5674:359–374, 2009.

Abstract: This paper reports on a case study, which we believe is the first to produce a formally verified end-to-end implementation of a functional programming language running on commercial processors. Interpreters for the core of McCarthys LISP 1.5 were implemented in ARM, x86 and PowerPC machine code, and proved to correctly parse, evaluate and print LISP s-expressions. The proof of evaluation required working on top of verified implementations of memory allocation and garbage collection. All proofs are mechanised in the HOL4 theorem prover.

[Myre09a] Magnus O. Myreen, Konrad Slind, and Michael J.C. Gordon. Extensible Proof-Producing Compilation. *LNCS*, 5501:2–16, 2009.

Abstract: This paper presents a compiler which produces machine code from functions defined in the logic of a theorem prover, and at the same time proves that the generated code executes the source functions. Unlike previously published work on proof-producing compilation from a theorem prover, our compiler provides broad support for user-defined extensions, targets multiple carefully modelled commercial machine languages, and does not require termination proofs for input functions. As a case study, the compiler is used to construct verified interpreters for a small LISP-like language. The compiler has been implemented in the HOL4 theorem prover.

[Myre10] Magnus O. Myreen. Verified Just-In-Time Compiler on x86. *ACM SIGLAN Notices - POPL'10*, 45(1):107–118, 2010.

> **Abstract:** This paper presents a method for creating formally correct just-in- time (JIT) compilers. The tractability of our approach is demonstrated through, what we believe is the first, verification of a JIT compiler with respect to a realistic semantics of self-modifying x86 machine code. Our semantics includes a model of the instruction cache. Two versions of the verified JIT compiler are presented: one generates all of the machine code at once, the other one is incremental i.e. produces code on-demand. All proofs have been performed inside the HOL4 theorem prover.

[Myre11] Magnus O. Myreen and Jared Davis. A Verified Runtime for a Verified Theorem Prover. *NCS*, 6898:265–280, 2011.

> **Abstract:** Theorem provers, such as ACL2, HOL, Isabelle and Coq, rely on the correctness of runtime systems for programming languages like ML, OCaml or Common Lisp. These runtime systems are complex and critical to the integrity of the theorem provers. In this paper, we present a new Lisp runtime which has been formally verified and can run the Milawa theorem prover. Our runtime consists of 7,500 lines of machine code and is able to complete a 4 gigabyte Milawa proof effort. When our runtime is used to carry out Milawa proofs, less unverified code must be trusted than with any other theorem prover. Our runtime includes a just-in-time compiler, a copying garbage collector, a parser and a printer, all of which are HOL4-verified down to the concrete x86 code. We make heavy use of our previously developed tools for machine-code verification. This work demonstrates that our approach to machine-code verification scales to non-trivial applications.

[Myre12] Magnus O. Myreen. Functional Programs: Conversions between Deep and Shallow Embeddings. *LNCS*, 7406:412–417, 2012.

> **Abstract:** This paper presents a method which simplifies verification of deeply embedded functional programs. We present a technique by which proof-certified equations describing the effect of functional programs (shallow embeddings) can be automatically extracted from their operational semantics. Our method can be used in reverse, i.e. from shallow to deep embeddings, and thus for implementing certifying code synthesis: we have implemented a tool which maps HOL functions to equivalent Lisp functions, for which we have a verified Lisp runtime. A key benefit, in both directions, is that the verifier does not need to understand the operational semantics that gives meanings to the deep embeddings.

[Myre14] Magnus O. Myreen and Jared Davis. The Reflective Milawa Theorem Prover is Sound. *LNAI*, pages 421–436, 2014.

> **Abstract:** Milawa is a theorem prover styled after ACL2 but with a small kernel and a powerful reflection mechanism. We

have used the HOL4 theorem prover to formalize the logic of Milawa, prove the logic sound, and prove that the source code for the Milawa kernel (2,000 lines of Lisp) is faithful to the logic. Going further, we have combined these results with our previous verification of an x86 machine-code implementation of a Lisp runtime. Our top-level HOL4 theorem states that when Milawa is run on top of our verified Lisp, it will only print theorem statements that are semantically true. We believe that this top-level theorem is the most comprehensive formal evidence of a theorem provers soundness to date.

[Neup13] Walther Neuper. Computer Algebra implemented in Isabelle's Function Package under Lucas-Interpretation – a Case Study, 2013.

Link: http://ceur-ws.org/Vol-1010/paper-09.pdf

[OCAM14] unknown. The OCAML website.

Link: http://ocaml.org

[Pfei12] Greg Pfeil. Common Lisp Type Hierarchy, 2012.

Link: http://sellout.github.io/2012/03/03/common-lisp-type-hierarchy

[Pier15] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catalin Hritcu, Vilhelm Sjöberg, 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.

[Pier17] Benjamin Pierce. DeepSpec Summer School, Coq Intensive, Part 1 (July 13,2017), 2017.

Link: https://www.youtube.com/watch?v=jG61w5pOc2A

[Reid17] Alastair Reid. How can you trust formally varified software, 2017.

Abstract: Formal verification of software has finally started to become viable: we have examples of formally verified microkernels, realistic compilers, hypervisors, etc. These are huge achievements and we can expect to see even more impressive results in the future but the correctness proofs depend on a number of assumptions about the Trusted Computing Base that the software depends on. Two key questions to ask are: Are the specifications of the Trusted Computing Base correct? And do the implementations match the specifications? I will explore the philosophical challenges and practical steps you can taek in answering that question for one of the major dependencies: the hardware your software runs on. I will describe the combination of formal verification and testing that ARM uses to verify the processor specification and I will talk about our current challenge: getting the specification down to zero bugs while the architecture continues to evolve.

Link: https://media.ccc.de/v/34c3-8915-how_can_you_trust_formally_verified_software#t=12

[Sant95] Philip S. Santas. A type system for computer algebra. *J. Symbolic Computation*, 19(1-3):79–109, 1995.

Abstract: This paper presents a type system for support of subtypes, parameterized types with sharing and categories in a computer algebra environment. By modeling representation of instances in terms of existential types, we obtain a simplified model, and build a basis for defining subtyping among algebraic domains. The inheritance at category level has been formalized; this allows the automatic inference of type classes. By means of type classes and existential types we construct subtype relations without involving coercions. A type sharing mechanism works in parallel and allows the consistent extension and combination of domains. The expressiveness of the system is further increased by viewing domain types as special case of package types, forming weak and strong sums respectively. The introduced system, although awkward at first sight, is simpler than other proposed systems for computer algebra without including some of their problems. The system can be further extended in other to support more constructs and increase its flexibility.

[Sho08] Victor Shoup. A Computational Introduction to Number Theory.

Link: http://shoup.net/ntb/ntb-v2.pdf

[Smol89a] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, Fachbereich Informatik, Universitat Kaiserslautern, 1989.

[Soze08] Mattieu Sozeau and Nicolas Oury. First-Class Type Classes. *Lecture Notes in Computer Science*, 5170:278–293, 2008.

Abstract: Type Classes have met a large success in Haskell and Isabelle, as a solution for sharing notations by overloading and for specifying with abstract structures by quantificaiton on contexts. However, both systems are limited by second-class implementations of these constructs, and these limitations are only overcome by ad-hoc extensions to the respective systems. We propose an embedding of type classes into a dependent type theory that is first-class and supports some of the most popular extensions right away. The implementation is correspondingly cheap, general, and integrates well inside the system, as we have experimented in Coq. We show how it can be used to help structured programming and proving by way of examples.

Link: https://www.irif.fr/~sozeau/research/publications/First-Class_Type_Classes.pdf

[Spit11] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Math. Struct. Comput. Sci.*, 21(4):795–825, 2011.

Abstract: The introduction of first-class type classes in the Coq system calls for a re-examination of the basic interfaces used for mathematical formalisation in type theory. We present a new set of type classes for mathematics and take full advantage of their unique features to make practical a particularly flexible approach that was formerly thought to be infeasible. Thus, we address traditional proof engineering challenges as well as new ones resulting from our ambition to build upon this development a library of constructive analysis in which any abstraction penalties inhibiting efficient computation are reduced to a minimum. The basis of our development consists of type classes representing a standard algebraic hierarchy, as well as portions of category theory and universal algebra. On this foundation, we build a set of mathematically sound abstract interfaces for different kinds of numbers, succinctly expressed using categorical language and universal algebra constructions. Strategic use of type classes lets us support these high-level theory-friendly definitions, while still enabling efficient implementations unhindered by gratuitous indirection, conversion or projection. Algebra thrives on the interplay between syntax and semantics. The Prolog-like abilities of type class instance resolution allow us to conveniently define a quote function, thus facilitating the use of reflective techniques.

Link: https://arxiv.org/pdf/1102.1323.pdf

[Stac17] StackExchange. How do Gap generate the elements in permutation groups, 2017.

Link: http://math.stackexchange.com/questions/1705277/how-do-gap-generate-the-elements-in-permutation-groups

[Suto87] Robert S. Sutor and Richard D. Jenks. The type inference and coercion facilities in the Scratchpad II interpreter. *SIGPLAN Notices*, 22(7):56–63, 1987, 0-89791-235-7.

Abstract: The Scratchpad II system is an abstract datatype programming language, a compiler for the language, a library of packages of polymorphic functions and parametrized abstract datatypes, and an interpreter that provides sophisticated type inference and coercion facilities. Although originally designed for the implementation of symbolic mathematical algorithms, Scratchpad II is a general purpose programming language . This paper discusses aspects of the implementation of the interpreter and how it attempts to provide a user friendly and relatively weakly typed front end for the strongly typed programming language.

Comment: IBM Research Report RC 12595 (#56575)

[Talp92] Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. In *Conf. on Logic in Computer Science*. Computer Science Press, 1992.

Abstract: The *type and effect discipline* is a new framework for reconstructing the principal type and the minimal effect of expressions in implicitly typed polymorphic functional languages that support imperative constructs. The type and effect discipline outperforms other polymorphic type systems. Just as types abstract collections of concrete values, *effects* denote imperative operations on regions. *Regions* abstract sets of possibly aliased memory locations. Effects are used to control type generalization in the presence of imperative constructs while regions delimit observable side-effects. The observable effects of an expression range over the regions that are free in its type environment and its type; effects related to local data structures can be discarded during type reconstruction. The type of an expression can be generalized with respect to the variables that are not free in the type environment or in the observable effect.

[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.

Link: http://www.ams.org/journals/bull/1994-30-02/S0273-0979-1994-00502-6/S0273-0979-1994-00502-6.pdf

[Tros13] Anne Trostle. An Algorithm for the Greatest Common Divisor, 2013.

Link: http://www.nuprl.org/MathLibrary/gcd/

[Wijn68] A. van Wijngarrden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.T. Meertens, and R.G. Fisker. Revised Report on the Algorithmic Language ALGOL 68, 1968.

Link: http://www.eah-jena.de/~kleine/history/languages/algol68-revisedreport.pdf

[Wiki14a] ProofWiki. Euclidean Algorithm.

Link: http://proofwiki.org/wiki/Euclidean_Algorithm

[Wiki14b] ProofWiki. Division Theorem.

Link: http://proofwiki.org/wiki/Division_Theorem

[Wiki17] Wikipedia. Calculus of constructions, 2017.

Link: https://en.wikipedia.org/wiki/Calculus_of_constructions

[WikiED] Wikipedia. Euclidean Domain, 2017.

Link: https://en.wikipedia.org/wiki/Euclidean_domain

[Wilk85a] Maurice Wilkes. *Memoirs of a Computer Pioneer*. MIT Press, 1985.

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

Link: http://www.cis.upenn.edu/~stevez/vellvm

Index