axiom™

The 30 Year Horizon

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

Volume 13: Proving Axiom Correct

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

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

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

Contents

Index **85**

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.

> **Writing is nature's way of letting you know how sloppy your thinking is** – Guindon[Lamp02]
>
> **Mathematics is nature's way of letting you know how sloppy your writing is.** – Leslie Lamport[Lamp02]
>
> **The existence of the computer is giving impetus to the discovery of algorithms that generate proofs. I can still hear the echos of the collective sigh of relief that greeted the announcement in 1970 that there is no general algorithm to test for integer solutions to polynomial Diophantine equations; Hilbert's tenth problem has no solution. Yet, as I look at my own field, I see that creating algorithms that generate proofs constitutes some of the most important mathematics being done. The all-purpose proof machine may be dead, but tightly targeted machines are thriving.** – Dave Bressoud [Bres93]
>
> **In contrast to humans, computers are good at performing formal processes. There are people working hard on the project of actually formalizing parts of mathematics by computer, with actual formally correct formal deductions. I think this is a very big but very worthwhile project, and I am confident that we will learn a lot from it. The process will help simplify and clarify mathematics. In not too many years, I expect that we will have interactive computer programs that can help people compile significant chunks of formally complete and correct mathematics (based on a few perhaps shaky but at least explicit assumptions) and that they will become part of the standard mathematician's working environment.** – William P. Thurston [Thur94]
>
> **Our basic premise is that the ability to construct and modify programs will not improve without a new and comprehensive look at the entire programming process. Past theoretical research, say, in the logic of programs, has tended to focus on methods for reasoning about individual programs; little has been done, it seems to us, to develop a sound understanding of the process of programming – the process by which programs evolve in concept and in practice. At present, we lack the means to describe the techniques of program construction and improvement in ways that properly link verification, documentation and adaptability.**
>
> – Scherlis and Scott (1983) in [Maso86]
>
> **...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]

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.

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]

Chapter 1

Here is a problem

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.

1.1 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.

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

1.2 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
```

1.3 Mathematics

From Buchberger[Buch97],
Define "divides"

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

Define "greatest common divisor"

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

Theorem:

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

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

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

Let t be arbitrary but fixed and assume

$$(t|a \wedge t|b) \tag{1.1}$$

We have to show

$$t|(a-b) \tag{1.2}$$

and

$$t|b \tag{1.3}$$

Equation 1.3 follows propositionally. For equation 1.2, by definition of "divides", we have to find a w such that

$$t \cdot w = a - b \tag{1.4}$$

From 1.1, 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

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

1.4 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.

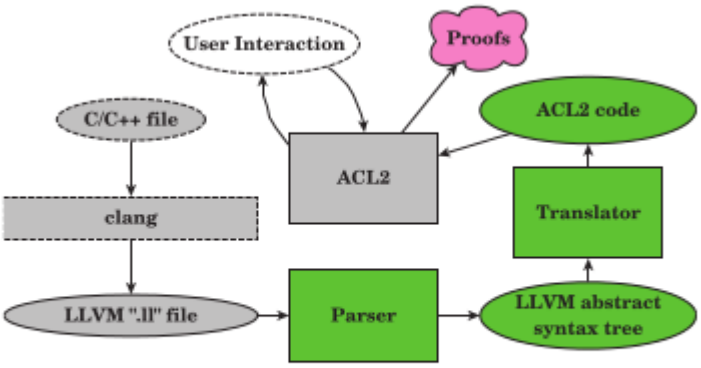


Figure 1: LLVM-to-ACL2 translation toolchain.

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 2

Theory

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

From WikiProof [Wiki14a] we quote:

Let

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

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

The steps of the algorithm are:

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

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

The proof is:

Suppose

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

and $aorb \neq 0$.

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

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

2.1 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

$$\begin{aligned} b &= aq_1 + r_1 \\ a &= r_1q_2 + r_2 \\ r_1 &= r_2q_3 + r_3 \\ &\vdots \\ r_{n-2} &= r_{n-1}q_n + r_n \\ r_{n-1} &= r_nq_{n+1} \end{aligned}$$

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

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

Chapter 3

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:

GCD(18;12;g)	=	GCD(12;18 rem 12;g)
	=	GCD(12;6;g
	=	GCD(6;12 rem 6;g)
	=	GCD(6;0;g)
	→	g = 6

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:

$\lambda n.letrcgcd(n) =$
$\lambda m.if n = 0 then m$
$else(gcd(mremn)n)$
$ingcd(n)$

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 4

Software Details

4.1 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 5

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.

5.1 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+.

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
```

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}
```

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
```

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
```

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
```

5.2 A simple proof

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]_`$<< x, y >>$

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").

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:

`BY` $e_1, \ldots, e_m$ `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
```

5.3 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 6

COQ proof of GCD

6.1 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

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.

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 $\mathbf{K}$ to mean either $\mathbf{P}$ or $\mathbf{T}$. 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

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

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 $x \notin \text{dom}(\Gamma)$
(weakening)	$\dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma, x : B \vdash M : A}$	if $x \notin \text{dom}(\Gamma)$
(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 $A =_\beta B$

Defining Logical Operators

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

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.

6.2 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.

6.3 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 7

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 8

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
```

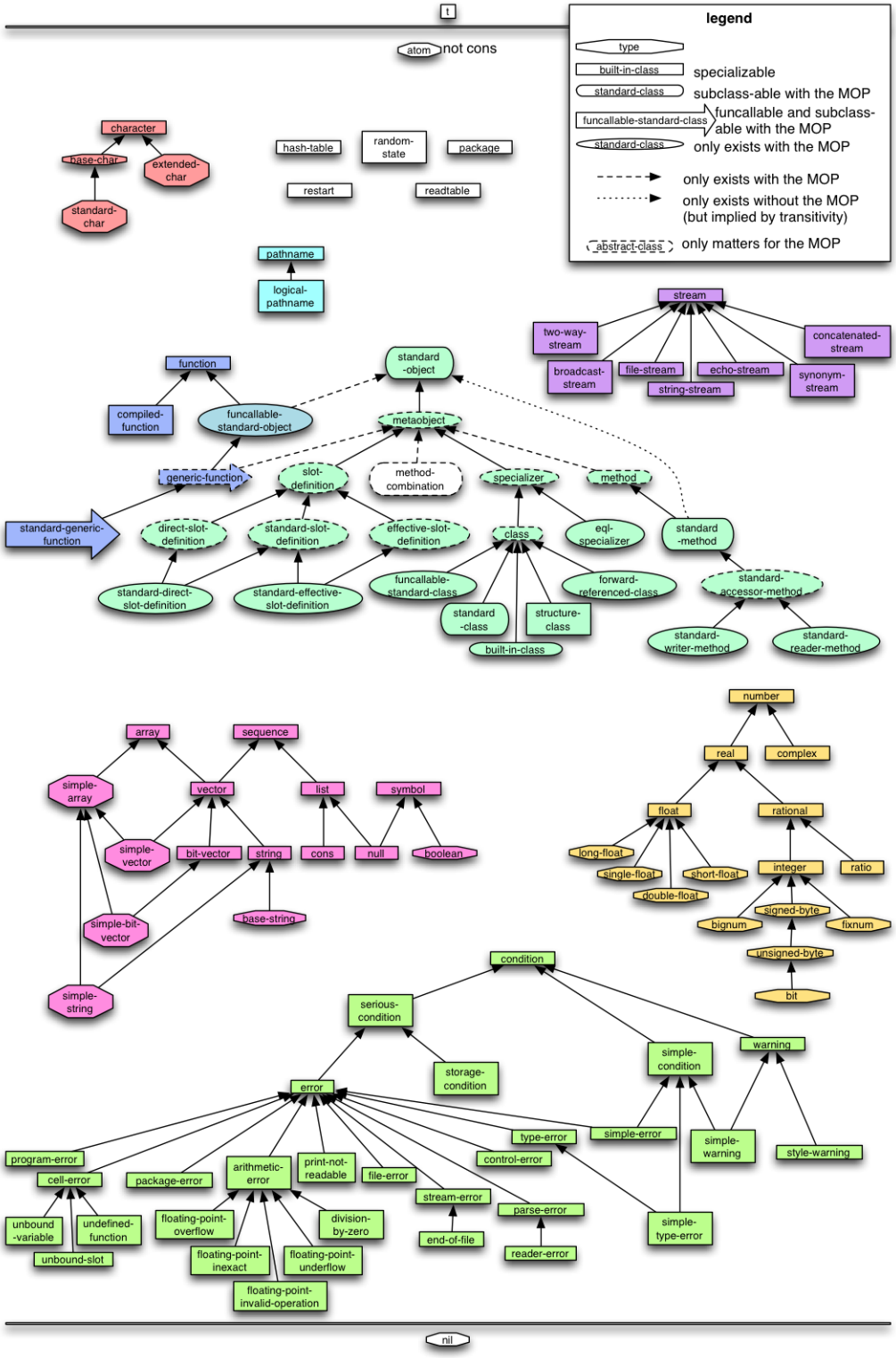
```
    @ variant n - i */
  while (i < n) {
    if (t[i] == v) break;
    i++;
  }
  return i;
}
```

Chapter 9

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 10

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

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 11

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.

11.1 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
```

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`.

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.

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.

11.2 More Monoids

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
```

11.3 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
```

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).
```

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.
```

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.
```

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
```

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 12

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 ⊨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 13

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

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

Adams[Adam01]

Ballarin[Ball95]

Davenport[Dave02]

Harrison[Harr98]

Clarke[Clar91] ... shows several proofs

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

[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 simpli-

fication 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

[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

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

[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

[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

[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 e 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

[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

[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

[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

[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

[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.

[Kreb17] Robbert Jan Krebbers. The ch$_2$o 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 Ssre-

flect: 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

[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.

[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.

[Sho08] Victor Shoup. A computational introduction to number theory.

Link: http://shoup.net/ntb/ntb-v2.pdf

[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

[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/

[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

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

Link: http://www.cis.upenn.edu/~stevez/vellvm

Index