# 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 10.1: Axiom Algebra: Theory


December 8, 2019

0ff38af175d383430dc97e8d643df0e8b76cd9f7

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

Paul Kosinski
Klaus Kusche
Larry Lambe
George L. Legendre
Michel Levaud
Ren-Cang Li
Craig Lucas
Camm Maguire
William Martin
Alasdair McAndrew
Bob McNeill
David Mentre
Gerard Milmeister
H. Michael Moeller
Scott Morrison
William Naylor
John Nelder
Jinzhong Niu
Kostas Oikonomou
Bill Page
Susan Pelzel
Ayal Pinkus
Jose Alfredo Portes
Beresford Parlett
Peter Poromaas
Arthur C. Ralfs
Guilherme Reis
Michael Richardson
Robert Risch
Simon Robinson
Martin Rubey
Philip Santas
William Schelter
Marshall Schor
Steven Segletes
William Sit
Matthieu Sozeau
Jonathan Steinbach
Klaus Sutner
Eugene Surowitz
T. Doug Telford
Balbir Thomas
Dylan Thurston
Dick Toupin
Hale Trotter
Kresimir Veselic
Bernhard Wall
Andreas Weber
Juergen Weiss
James Wen
R. Clint Whaley
Berhard Will
Shmuel Winograd
Waldemar Wiwianka
Liu Xiaojun
Qian Yun
Evelyn Zoernack

Igor Kozachenko
Bernhard Kutzler
Kaj Laurson
Franz Lehner
Howard Levy
John Lipson
Michael Lucks
Dave Mainey
Ursula Martin
Bob McElrath
Edi Meier
Jonathan Millen
William Miranker
Michael Monagan
Joel Moses
Patrice Naudin
Godfrey Nolan
Michael O'Connor
Humberto Ortiz-Zuazaga
David Parnas
Michel Petitot
Frederick H. Pitts
E. Quintana-Orti
A. Petitet
Greg Puhak
Norman Ramsey
Huan Ren
Jason Riedy
Jean Rivlin
Raymond Rogers
Jeff Rutter
David Saunders
Gerhard Schneider
Frithjof Schulze
V. Sima
Elena Smirnova
Srinivasan Seshan
Fabio Stumbo
Robert Sutor
Yong Kiam Tan
James Thatcher
Mike Thomas
Francoise Tisseur
Raymond Toy
Themos T. Tsikas
Christof Voemel
Paul Wang
Jaap Weel
M. Weller
Thorsten Werther
James T. Wheeler
Clifton J. Williamson
Robert Wisbauer
Knut Wolf
Clifford Yapp
Vadim Zhytnikov
Bruno Zuercher

Fred Krogh
Tim Lahey
Charles Lawson
Frederic Lehobey
J. Lewis
Rudiger Loos
Richard Luczak
Francois Maltey
Osni Marques
Michael McGettrick
Ian Meikle
Victor S. Miller
Mohammed Mobarak
Marc Moreno-Maza
Mark Murray
C. Andrew Neff
Arthur Norman
Summat Oemrawsingh
Julian A. Padget
Norm Pass
Didier Pinchon
Frank Pfenning
Gregorio Quintana-Orti
Andre Platzer
Claude Quitte
Anatoly Raportirenko
Albert D. Rich
Renaud Rioboo
Nicolas Robidoux
Michael Rothstein
R.W Ryniker II
Alfred Scheerhorn
Martin Schoenert
Fritz Schwartz
Nick Simicich
Jacob Nyffeler Smith
Ken Stanley
Christine Sundaresan
Moss E. Sweedler
Max Tegmark
Laurent Thery
Carol Thompson
Steve Toleque
Barry Trager
Gregory Vanuxem
E.G. Wagner
Stephen Watt
Al Weis
Mark Wegman
Michael Wester
John M. Wiley
Stephen Wilson
Sandra Wityak
Yanyang Xiao
David Yun
Richard Zippel
Dan Zwillinger

# Contents

# New Foreword

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

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

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

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

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

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

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

# Chapter 1

# Interval Arithmetic

Lambov [Lamb06] defines a set of useful formulas for computing intervals using the IEEE-754 floating-point standard.

The first thing to note is that IEEE floating point defaults to **round-to-nearest**. However, Lambov sets the rounding mode to **round to** $-\infty$. Computing lower bounds directly uses the hardware floating point operations but computing upper bounds he uses the identity

$$\Delta(x) = -\nabla(-x)$$

so that the upper bound of the pair of bounds is always negated. That is,

$$x = [\underline{x}, \overline{x}] = \langle \underline{x}, -\overline{x} \rangle$$

Given that convention

- the sum of $x$ and $y$ is evaluated by

$$\langle \nabla(\underline{x} + \underline{y}), -\nabla(-\overline{x} - \overline{y}) \rangle$$

- changing the sign of an interval $x$ is achieved by swapping the two bounds, that is $\langle -\overline{x}, \underline{x} \rangle$
- joining two intervals (that is, finding an interval containing all numbers in both, or finding the minimum of the lower bounds and the maximum of the higher bounds) is performed as

$$\langle min(\underline{x}, \underline{y}), -min((-\overline{x}), (-\overline{y})) \rangle$$

Lambov defines operations which, under the given rounding condition, give the tightest bounds.

## 1.1 Addition

$$x + y = \left[\underline{x} + \underline{y}, \overline{x} + \overline{y}\right] \subseteq \left\langle \nabla(\underline{x} + \underline{y}) - \nabla((-\overline{x}) + (-\overline{y})) \right\rangle$$

The negated sign of the higher bound ensures the proper direction of the rounding.

## 1.2 Sign Change

$$-x = [-\overline{x}, -\underline{x}] = \langle -\overline{x}, \underline{x} \rangle$$

This is a single swap of the two values. No rounding is performed.

## 1.3 Subtraction

$$x - y = \left[\underline{x} - \overline{y}, \overline{x} - \underline{y}\right] \subseteq \left\langle \nabla(\underline{x} + (-\overline{y})), -\nabla((-\overline{x}) + \underline{y}) \right\rangle$$

Subtraction is implemented as $x + (-y)$.

## 1.4 Multiplication

$$xy = \left[min(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}), max(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y})\right]$$

The rounding steps are part of the operation so all 8 multiplications are required. Lambov notes that since

$$\Delta(\nabla(r) + \epsilon) \geq \Delta(r)$$

for $\epsilon$ being the smallest representable positive number, one can do with 4 multiplications at the expense of some accuracy.

In Lambov's case he makes the observation that

$$xy = \begin{cases} \left[min(\underline{x}\underline{y}, \overline{x}\underline{y}), max(\overline{x}\overline{y}, \underline{x}\overline{y})\right], & \text{if } 0 \leq \underline{x} \leq \overline{x} \\ \left[min(\underline{x}\overline{y}, \overline{x}\underline{y}), max(\overline{x}\overline{y}, \underline{x}\underline{y})\right], & \text{if } \underline{x} < 0 \leq \overline{x} \\ \left[min(\underline{x}\overline{y}, \overline{x}\overline{y}), max(\overline{x}\underline{y}, \underline{x}\underline{y})\right], & \text{if } \underline{x} \leq \overline{x} < 0 \end{cases}$$

from which he derives the formula actually used

$$xy \subseteq \langle min(\nabla(a\underline{x}), \nabla(b(-\overline{x}))), -min(\nabla(c(-\overline{x})), \nabla(d\underline{x})) \rangle$$

where

$$\begin{aligned} a &= \begin{cases} \underline{y} & \text{if } 0 \leq \underline{x} \\ -(-\overline{y}) & \text{otherwise} \end{cases} \\ b &= \begin{cases} -\underline{y} & \text{if } (-\overline{x}) \leq 0 \\ (-\overline{y}) & \text{otherwise} \end{cases} \\ c &= \begin{cases} -(-\overline{y}) & \text{if } (-\overline{x}) \leq 0 \\ \underline{y} & \text{otherwise} \end{cases} \\ d &= \begin{cases} (-\overline{y}) & \text{if } 0 \leq \underline{x} \\ -\underline{y} & \text{otherwise} \end{cases} \end{aligned}$$

which computes the rounded results of the original multiplication formula but achieves better performance.

## 1.5 Multiplication by a positive number

If one of the numbers is known to be positive (e.g. a constant) then

$$\text{if } x > 0 \text{ then } xy \equiv \left[min(\underline{x}\underline{y}, \overline{x}\underline{y}), max(\overline{x}\overline{y}, \underline{x}\overline{y})\right]$$

This formula is faster than the general multiplication formula.

## 1.6 Multiplication of Two Positive Numbers

If both multiples are positive simply change the sign of the higher bound on one of the arguments prior to multiplication. If one of the numbers is a constant this can be arranged to skip the sign change.

## 1.7 Division

Division is an expensive operation.

$$\frac{x}{y} = \left[ min\left(\frac{\underline{x}}{\underline{y}}, \frac{\underline{y}}{\overline{y}}, \frac{\overline{x}}{\underline{y}}, \frac{\overline{x}}{\overline{y}}\right), max\left(\frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\overline{y}}, \frac{\overline{x}}{\underline{y}}, \frac{\overline{x}}{\overline{y}}\right)\right]$$

which is undefined if $0 \in y$. To speed up the computation Lambov uses the identity

$$\frac{x}{y} = x\frac{1}{y}$$

Lambov does a similar analysis to improve the overall efficiency.

$$\frac{x}{y} = \begin{cases} \left[min(\frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\overline{y}}), max(\frac{\overline{x}}{\underline{y}}, \frac{\overline{x}}{\overline{y}})\right], & \text{if } 0 < \underline{y} \leq \overline{y} \\ \text{exception} & \text{if } \underline{y} \leq 0 \leq \overline{y} \\ \left[min(\frac{\overline{x}}{\underline{y}}, \frac{\overline{x}}{\overline{y}}), max(\frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\overline{y}})\right], & \text{if } \underline{y} \leq \overline{y} \leq 0 \end{cases}$$

The formula he uses is

$$\frac{x}{y} \subseteq \left\langle min\left(\nabla\left(\frac{a}{\underline{y}}\right), \nabla\left(\frac{-a}{(-\overline{y})}\right)\right), -min\left(\nabla\left(\frac{-b}{(-\overline{y})}\right), \nabla\left(\frac{b}{\underline{y}}\right)\right)\right\rangle$$

where

$$\begin{aligned} a &= \begin{cases} \underline{x} & \text{if } (-\overline{y}) \leq 0 \\ -(-\overline{x}) & \text{otherwise} \end{cases} \\ b &= \begin{cases} (-\overline{x}) & \text{if } 0 \leq \underline{y} \\ -\underline{x} & \text{otherwise} \end{cases} \end{aligned}$$

## 1.8 Reciprocal

$$\frac{1}{x} = \left[\frac{1}{\overline{x}}, \frac{1}{\underline{x}}\right] \subseteq \left\langle \nabla\left(\frac{-1}{(-\overline{x})}\right), \nabla\left(\frac{-1}{\underline{x}}\right)\right\rangle$$

which is undefined if $0 \in x$. Lambov implements this by checking for zero, followed by division of $-1$ by the argument and swapping the two components.

## 1.9 Absolute Value

$$|x| = [max(\underline{x}, -\overline{x}, 0), max(-\underline{x}, \overline{x})] = \langle max(0, \underline{x}, (-\overline{x})), -min(\underline{x}, (-\overline{x}))\rangle$$

## 1.10 Square

$$x^2 = |x|\,|x|$$

using multiplication by positive numbers, mentioned above.

## 1.11 Square Root

$$\sqrt{x} = \left[\sqrt{\underline{x}}, \sqrt{\overline{x}}\right]$$

which is defined if $0 \leq \underline{x}$

Lambov notes that this formula has a rounding issue. He notes that since

$$\Delta(r) \leq -\nabla(-\epsilon - \nabla(r))$$

he uses the formula

$$\sqrt{x} \subseteq \begin{cases} \left\langle \nabla\left(\sqrt{\underline{x}}\right), -\nabla\left(\sqrt{-(-\overline{x})}\right)\right\rangle, \text{ if } \nabla\left(\nabla\left(\sqrt{-(-\overline{x})}\right)\right)^2 = -(-\overline{x}) \\ \left\langle \nabla\left(\sqrt{\underline{x}}\right), \nabla\left(\nabla\left(-\epsilon - \sqrt{-(-\overline{x})}\right)\right)\right\rangle, \text{ otherwise} \end{cases}$$

where $\epsilon$ is the smallest representable positive number.

The first branch of this formula is only satisfied if the result of $\sqrt{-(-\overline{x})}$ is exactly representable, in which case

$$\nabla\left(\sqrt{-(-\overline{x})}\right) = \nabla\left(\sqrt{-(-\overline{x})}\right)$$

otherwise the second branch of the formula adjusts the high bound to the next representable number. If tight bounds are not required the second branch is always sufficient.

If the argument is entirely negative, the implementation will raise an exception. If it contains a negative part, the implementation will crop it to only its non-negative part to allow that computations such as $\sqrt{0}$ ca be carried out in exact real arithmetic.

# Chapter 2

# Integration

An *elementary function*[Bro98b] of a variable $x$ is a function that can be obtained from the rational functions in $x$ by repeatedly adjoining a finite number of nested logarithms, exponentials, and algebraic numbers or functions. Since $\sqrt{-1}$ is elementary, the trigonometric functions and their inverses are also elementary (when they are rewritten using complex exponentials and logarithms) as well as all the "usual" functions of calculus. For example,

$$\sin(x + \tan(x^3 - \sqrt{x^3 - x + 1})) \tag{2.1}$$

is elementary when rewritten as

$$\frac{\sqrt{-1}}{2}(e^{t-x\sqrt{-1}} - e^{x\sqrt{-1}-t}) \text{ where } t = \frac{1 - e^{2\sqrt{-1}(x^3-\sqrt{x^3-x+1})}}{1 + e^{2\sqrt{-1}(x^3-\sqrt{x^3-x+1})}}$$

This tutorial describes recent algorithmic solutions to the *problem of integration in finite terms*: to decide in a finite number of steps whether a given elementary funcction has an elementary indefinite integral, and to compute it explicitly if it exists. While this problem was studied extensively by Abel and Liouville during the last century, the difficulties posed by algebraic functions caused Hardy (1916) to state that "there is reason to suppose that no such method can be given". This conjecture was eventually disproved by Risch (1970), who described an algorithm for this problem in a series of reports [Ostr1845, Risc68, Risc69a, Risc69b, Risc70]. In the past 30 years, this procedure has been repeatedly improved, extended and refined, yielding practical algorithms that are now becoming standard and are implemented in most of the major computer algebra systems. In this tutorial, we outline the above algorithms for various classes of elementary functions, starting with rational functions and progressively increasing the class of functions up to general elementary functions. Proofs of correctness of the algorithms presented here can be found in several of the references, and are generally too long and too detailed to be described in this tutorial.

**Notations**: we write $x$ for the variable of integration, and $'$ for the derivation $d/dx$. $\mathbb{Z}$,$\mathbb{Q}$,$\mathbb{R}$,and $\mathbb{C}$ denote respectively the integers, rational, real and complex numbers. All fields are commutative and, except when mentioned explicitly otherwise, have characteristic 0. If $K$ is a field, then $\overline{K}$ denotes its algebraic closure. For a polynomial $p$, $\mathrm{pp}(p)$ denotes the primitive part of $p$, *i. e.* $p$ divided by the gcd of its coefficients.

## 2.1 Rational Functions

By a *rational function*, we mean a quotient of polynomials in the integration variable $x$. This means that other functions can appear in the integrand, provided they do not involve $x$, hence that the coefficients of our polynomials in $x$ lie in an arbitrary field $K$ satisfying: $\forall a \in K,\ a' = 0$.

### 2.1.1 The full partial-fraction algorithm

This method, which dates back to Newton, Leibniz, and Bernoulli, should not be used in practice, yet it remains the method found in most calculus tests and is often taught. Its major drawback is the factorization of the denominator of the integrand over the real or complex numbers. We outline it because it provides the theoretical foundations for all the subsequent algorithms. Let $f \in \mathbb{R}(x)$ be our integrand, and write $f = P + A/D$ where $P, A, D \in \mathbb{R}[x]$, $\gcd(A, D) = 1$, and $\deg(A) < \deg(D)$. Let

$$D = c\prod_{i=1}^{n}(x - a_i)^{e_i}\prod_{j=1}^{m}(x^2 + b_j x + c_j)^{f_j}$$

be the irreducible factorization of $D$ over $\mathbb{R}$, where $c$, the $a_i$'s, $b_j$'s and $c_j$'s are in $\mathbb{R}$ and the $e_i$'s and $f_j$'s are positive integers. Computing the partial fraction decomposition of $f$, we get

$$f = P + \sum_{i=1}^{n}\sum_{k=1}^{e_i}\frac{A_{ik}}{(x - a_i)^k} + \sum_{j=1}^{m}\sum_{k=1}^{f_i}\frac{B_{jk}x + C_{jk}}{(x^2 + b_j x + c_j)^k}$$

where the $A_{ik}$'s, $B_{jk}$'s, and $C_{jk}$'s are in $\mathbb{R}$. Hence,

$$\int f = \int P + \sum_{i=1}^{n}\sum_{k=1}^{e_i}\int\frac{A_{ik}}{(x - a_i)^k} + \sum_{j=1}^{m}\sum_{k=1}^{f_i}\int\frac{B_{jk}x + C_{jk}}{(x^2 + b_j x + c_j)^k}$$

Computing $\int P$ poses no problem (it will for any other class of functions), and for the other terms we have

$$\int\frac{A_{ik}}{(x - a_i)^k} = \begin{cases} A_{ik}(x - a_i)^{1-k}/(1 - k) & \text{if } k > 1 \\ A_{i1}\log(x - a_i) & \text{if } k = 1 \end{cases} \tag{2.2}$$

and, noting that $b_j^2 - 4c_j < 0$ since $x^2 + b_j x + c_j$ is irreducible in $\mathbb{R}$[x].

$$\int\frac{B_{j1}x + C_{j1}}{(x^2 + b_j x + c_j)} = \frac{B_{j1}}{2}\log(x^2 + b_j x + c_j) + \frac{2C_{j1} - b_j B_{j1}}{\sqrt{4c_j - b_j^2}}arctan\left(\frac{2x + b_j}{\sqrt{4c_j - b_j^2}}\right)$$

and for $k > 1$,

$$\begin{aligned}\int\frac{B_{jk}x + C_{jk}}{(x^2 + b_j x + c_j)^k} &= \frac{(2C_{jk} - b_j B_{jk})x + b_j C_{jk} - 2c_j B_{jk}}{(k - 1)(4c_j - b_j^2)(x^2 + b_j x + c_j)^{k-1}} \\ &\quad + \int\frac{(2k - 3)(2C_{jk} - b_j B_{jk})}{(k - 1)(4c_j - b_j^2)(x^2 + b_j x + c_j)^{k-1}}\end{aligned}$$

This last formula is then used recursively until $k = 1$.

An alternative is to factor $D$ linearly over $\mathbb{C}$: $D = \prod_{i=1}^{q}(x-\alpha_i)^{e_i}$, and then use 2.2 on each term of

$$f = P + \sum_{i=1}^{q}\sum_{j=1}^{e_i}\frac{A_{ij}}{(x-\alpha_i)^j} \tag{2.3}$$

Note that this alternative is applicable to coefficients in any field $K$, if we factor $D$ linearly over its algebraic closure $\overline{K}$, and is equivalent to expanding $f$ into its Laurent series at all its finite poles, since that series at $x = \alpha_i \in \overline{K}$ is

$$f = \frac{A_{ie_i}}{(x-\alpha_i)^{e_i}} + \cdots + \frac{A_{i2}}{(x-\alpha_i)^2} + \frac{A_{i1}}{(x-\alpha_i)} + \cdots$$

where the $A_{ij}$'s are the same as those in 2.3. Thus, this approach can be seen as expanding the integrand into series around all the poles (including $\infty$), then integrating the series termwise, and then interpolating for the answer, by summing all the polar terms, obtaining the integral of 2.3. In addition, this alternative shows that any rational function $f \in K(x)$ has an elementary integral of the form

$$\int f = v + c_1\log(u_1) + \cdots + c_m\log(u_m) \tag{2.4}$$

where $v, u_1, \ldots, u_m \in \overline{K}(x)$ are the rational functions, and $c_1, \ldots, c_m \in \overline{K}$ are constants. The original Risch algorithm is essentially a generalization of this approach that searches for integrals of arbitrary elementary functions in a form similar to 2.4.

### 2.1.2 The Hermite reduction

The major computational inconvenience of the full partial fraction approach is the need to factor polynomials over $\mathbb{R}$, $\mathbb{C}$, or $\overline{K}$, thereby introducing algebraic numbers even if the integrand and its integral are both in $\mathbb{Q}(x)$. On the other hand, introducing algebraic numbers may be necessary, for example it is proven in [Risc69a] that any field containing an integral of $1/(x^2+2)$ must also contain $\sqrt{2}$. Modern research has yielded so-called "rational" algorithms that

- compute as much of the integral as possible with all calculations being done in $K(x)$, and
- compute the minimal algebraic extension of $K$ necessary to express the integral

The first rational algorithms for integration date back to the 19$^{\text{th}}$ century, when both Hermite [Herm1872] and Ostrogradsky [Ostr1845] invented methods for computing the $v$ of 2.4 entirely within $K(x)$. We describe here only Hermite's method, since it is the one that has been generalized to arbitrary elementary functions. The basic idea is that if an irreducible $p \in K[x]$ appears with multiplicity $k > 1$ in the factorization of the denominator of the integrand, then 2.2 implies that it appears with multiplicity $k-1$ in the denominator of the integral. Furthermore, it is possible to compute the product of all such irreducibles for each $k$ without factoring the denominator into irreducibles by computing its *squarefree factorization*, *i.e* a factorization $D = D_1D_2^2\cdots D_m^m$, where each $D_i$ is squarefree and $\gcd(D_i, D_j) = 1$ for $i \neq j$. A straightforward way to compute it is as follows: let $R = \gcd(D, D')$, then $R = D_2D_2^3\cdots D_m^{m-1}$, so $D/R = D_1D_2\cdots D_m$ and $\gcd(R, D/R) = D_2\cdots D_m$, which implies finally that

$$D_1 = \frac{D/R}{\gcd(R, D/R)}$$

Computing recursively a squarefree factorization of $R$ completes the one for $D$. Note that [Yun76] presents a more efficient method for this decomposition. Let now $f \in K(x)$ be our integrand, and write $f = P + A/D$ where $P, A, D \in K[x]$, $\gcd(A, D) = 1$, and $\deg(A) < \deg(D)$. Let $D = D_1 D_2^2 \cdots D_m^m$ be a squarefree factorization of $D$ and suppose that $m \geq 2$ (otherwise $D$ is already squarefree). Let then $V = D_m$ and $U = D/V^m$. Since $\gcd(UV', V) = 1$, we can use the extended Euclidean algorithm to find $B, C \in K[x]$ such that

$$\frac{A}{1-m} = BUV' + CV$$

and $\deg(B) < \deg(V)$. Multiplying both sides by $(1-m)/(UV^m)$ gives

$$\frac{A}{UV^m} = \frac{(1-m)BV'}{V^m} + \frac{(1-m)C}{UV^{m-1}}$$

so, adding and subtracting $B'/V^{m-1}$ to the right hand side, we get

$$\frac{A}{UV^m} = \left(\frac{B'}{V^{m-1}} - \frac{(m-1)BV'}{V^m}\right) + \frac{(1-m)C - UB'}{UV^{m-1}}$$

and integrating both sides yields

$$\int \frac{A}{UV^m} = \frac{B}{V^{m-1}} + \int \frac{(1-m)C - UB'}{UV^{m-1}}$$

so the integrand is reduced to one with a smaller power of $V$ in the denominator. This process is repeated until the denominator is squarefree, yielding $g, h \in K(x)$ such that $f = g' + h$ and $h$ has a squarefree denominator.

### 2.1.3 The Rothstein-Trager and Lazard-Rioboo-Trager algorithms

Following the Hermite reduction, we only have to integrate fractions of the form $f = A/D$ with $\deg(A) < \deg(D)$ and $D$ squarefree. It follows from 2.2 that

$$\int f = \sum_{i=1}^{n} a_i \log(x - \alpha_i)$$

where the $\alpha_i$'s are the zeros of $D$ in $\overline{K}$, and the $a_i$'s are the residues of $f$ at the $\alpha_i$'s. The problem is then to compute those residues without splitting $D$. Rothstein [Roth77] and Trager [Trag76] independently proved that the $\alpha_i$'s are exactly the zeros of

$$R = \text{resultant}_x(D, A - tD') \in K[t] \tag{2.5}$$

and that the splitting field of $R$ over $K$ is indeed the minimal algebraic extension of $K$ necessary to express the integral in the form 2.4. The integral is then given by

$$\int \frac{A}{D} = \sum_{i=1}^{m} \sum_{a | R_i(a) = 0} a \log(\gcd(D, A - aD')) \tag{2.6}$$

where $R = \prod_{i=1}^{m} R_i^{e_i}$ is the irreducible factorization of $R$ over $K$. Note that this algorithm requires factoring $R$ into irreducibles over $K$, and computing greatest common divisors in $(K[t]/(R_i))[x]$, hence computing with algebraic numbers. Trager and Lazard & Rioboo

[Laza90] independently discovered that those computations can be avoided, if one uses the subresultant PRS algorithm to compute the resultant of 2.5: let $(R_0, R_1, \ldots R_k \neq 0, 0, \ldots)$ be the subresultant PRS with respect to $x$ of $D$ and $A - tD'$ and $R = Q_1 Q_2^2 \ldots Q_m^m$ be a *squarefree* factorization of their resultant. Then,

$$\sum_{a|Q_i(a)=0} a \log(\gcd(D, A - aD')) =$$

$$\begin{cases} \sum_{a|Q_i(a)=0} a \log(D) & \text{if } i = \deg(D) \\ \sum_{a|Q_i(a)=0} a \log(\mathrm{pp}_x(R_{k_i})(a, x)) & \text{where } \deg(R_{k_i}) = i, 1 \leq k_i \leq n \\ & \text{if } i < \deg(D) \end{cases}$$

Evaluating $\mathrm{pp}_x(R_{k_i})$ at $t = a$ where $a$ is a root of $Q_i$ is equivalent to reducing each coefficient with respect to $x$ of $\mathrm{pp}_x(R_{k_i})$ module $Q_i$, hence computing in the algebraic extension $K[t]/(Q_i)$. Even this step can be avoided: it is in fact sufficient to ensure that $Q_i$ and the leading coefficient with respect to $x$ of $R_{k_i}$ do not have a nontrivial common factor, which implies then that the remainder by $Q_i$ is nonzero, see [Muld97] for details and other alternatives for computing $\mathrm{pp}_x(R_{k_i})(a, x)$

## 2.2 Algebraic Functions

By an *algebraic function*, we mean an element of a finitely generated algebraic extension $E$ of the rational function field $K(x)$. This includes nested radicals and implicit algebraic functions, not all of which can be expressed by radicals. It turns out that the algorithms we used for rational functions can be extended to algebraic functions, but with several difficulties, the first one being to define the proper analogues of polynomials, numerators and denominators. Since $E$ is algebraic over $K(x)$, for any $\alpha \in E$, there exists a polynomial $p \in K[x][y]$ such that $p(x, \alpha) = 0$. We say that $\alpha \in E$ is *integral over* $K[x]$ if there is a polynomial $p \in K[x][y]$, *monic in* $y$, such that $p(x, \alpha) = 0$. Integral elements are analogous to polynomials in that their value is defined for any $x \in \overline{K}$ (unlike non-integral elements, which must have at least one pole in $\overline{K}$). The set

$$\mathbf{O}_{K[x]} = \{\alpha \in E \text{ such that } \alpha \text{ is integral over } K[x]\}$$

is called the *integral closure of* $K[x]$ *in* $E$. It is a ring and a finitely generated $K[x]$-module. Let $\alpha \in E^*$ be any element and $p = \sum_{i=0}^m a_i y^i \in K[x][y]$ be such that $p(x, \alpha) = 0$ and $a_m \neq 0$. Then, $q(x, a_m y) = 0$ where $q = y^m + \sum_{i=0}^{m-1} a_i a_m^{m-i-1} y^i$ is monic in $y$, so $a_m y \in \mathbf{O}_{K[x]}$. We need a canonical representation for algebraic functions similar to quotients of polynomials for rational functions. Expressions as quotients of integral functions are not unique, for example, $\sqrt{x}/x = x/\sqrt{x}$. However, $E$ is a finite-dimensional vector space over $K(x)$, so let $n = [E : K(x)]$ and $w = (w_1, \ldots, w_n)$ be any basis for $E$ over $K(x)$. By the above remark, there are $a_1, \ldots, a_n \in K(x)^*$ such that $a_i w_i \in \mathbf{O}_{K[x]}$ for each $i$. Since $(a_1 w_1, \ldots, a_n w_n)$ is also a basis for $E$ over $K(x)$, we can assume without loss of generality that the basis $w$ is composed of integral elements. Any $\alpha \in E$ can be written uniquely as $\alpha = \sum_{i=1}^n f_i w_i$ for $f_1, \ldots, f_n \in K(x)$, and putting the $f_i$'s over a monic common denominator $D \in K[x]$, we get an expression

$$\alpha = \frac{A_1 w_1 + \ldots + A_n w_n}{D}$$

where $A_1, \ldots, A_n \in K[x]$ and $\gcd(D, A_1, \ldots, A_n) = 1$. We call $\sum_{i=1}^n A_i w_i \in \mathbf{O}_{K[x]}$ and $D \in K[x]$ respectively the *numerator* and *denominator* of $\alpha$ with respect to $w$. They are defined uniquely once the basis $w$ is fixed.

### 2.2.1 The Hermite reduction

Now that we have numerators and denominators for algebraic functions, we can attempt to generalize the Hermite reduction of the previous section, so let $f \in E$ be our integrand, $w = (w_1, \ldots, w_n) \in \mathbf{O}_{K[n]}{}^n$ be a basis for $E$ over $K(x)$ and let $\sum_{i=1}^m A_i w_i \in \mathbf{O}_{K[x]}$ and $D \in K[x]$ be the numerator and denominator of $f$ with respect to $w$, Let $D = D_1 D_2^2 \ldots D_m^m$ be a squarefree factorization of $D$ and suppose that $m \geq 2$. Let then $V = D_m$ and $U = D/V^m$, and we ask whether we can compute $B = \sum_{i=1}^n B_i w_i \in \mathbf{O}_{K[x]}$ and $h \in E$ such that $\deg(B_i) < \deg(V)$ for each $i$,

$$\int \frac{\sum_{i=1}^n A_i w_i}{UV^m} = \frac{B}{V^{m-1}} + \int h \tag{2.7}$$

and the denominator of $h$ with respect to $w$ has no factor of order $m$ or higher. This turns out to reduce to solving the following linear system

$$f_1 S_1 + \ldots + f_n S_n = A_1 w_1 + \ldots + A_n w_n \tag{2.8}$$

for $f_1, \ldots, f_n \in K(x)$, where

$$S_i = UV^m \left(\frac{w_i}{V^{m-1}}\right)' \quad \text{for } 1 \leq i \leq n \tag{2.9}$$

Indeed, suppose that 2.8 has a solution $f_1, \ldots, f_n \in K(x)$, and write $f_i = T_i/Q$, where $Q, T_1, \ldots, T_n \in K[x]$ and $\gcd(Q, T_1, \ldots, T_n) = 1$. Suppose further that $\gcd(Q, V) = 1$. Then, we can use the extended Euclidean algorithm to find $A, R \in K[x]$ such that $AV + RQ = 1$, and Euclidean division to find $Q_i, B_i \in K[x]$ such that $\deg(B_i) < \deg(V)$ when $B_i \neq 0$ and $RT_i = VQ_i + B_i$ for each $i$. We then have

$$\begin{aligned}
h &= f - \left(\frac{\sum_{i=1}^n B_i w_i}{V^{m-1}}\right)' \\
&= \frac{\sum_{i=1}^n A_i w_i}{UV^m} - \frac{\sum_{i=1}^n B_i' w_i}{V^{m-1}} - \sum_{i=1}^n (RT_i - VQ_i)\left(\frac{w_i}{V^{m-1}}\right)' \\
&= \frac{\sum_{i=1}^n A_i w_i}{UV^m} - \frac{R\sum_{i=1}^n T_i S_i}{UV^m} + V\sum_{i=1}^n Q_i \left(\frac{w_i}{V^{m-1}}\right)' - \frac{\sum_{i=1}^n B_i' w_i}{V^{m-1}} \\
&= \frac{(1-RQ)\sum_{i=1}^n A_i w_i}{UV^m} + \frac{\sum_{i=1}^n Q_i w_i'}{V^{m-2}} - (m-1)V'\frac{\sum_{i=1}^n Q_i w_i}{V^{m-1}} - \frac{\sum_{i=1}^n B_i' w_i}{V^{m-1}} \\
&= \frac{\sum_{i=1}^n AA_i w_i}{UV^{m-1}} - \frac{\sum_{i=1}^n ((m-1)V'Q_i + B_i')w_i}{V^{m-1}} + \frac{\sum_{i=1}^n Q_i w_i'}{V^{m-2}}
\end{aligned}$$

Hence, if in addition the denominator of $h$ has no factor of order $m$ or higher, then $B = \sum_{i=1}^n B_i w_i \in \mathbf{O}_{K[x]}$ and $h$ solve 2.7 and we have reduced the integrand. Unfortunately, it can happen that the denominator of $h$ has a factor of order $m$ or higher, or that 2.8 has no solution in $K(x)$ whose denominator is coprime with $V$, as the following example shows.

**Example 1** Let $E = K(x)[y]/(y^4 + (x^2 + x)y - x^2)$ *with basis* $w = (1, y, y^2, y^3)$ *over* $K(x)$ *and consider the integrand*

$$f = \frac{y^3}{x^2} = \frac{w_4}{x^2} \in E$$

We have $D = x^2$, so $U = 1, V = x$ and $m = 2$. Then, $S_1 = x^2(1/x)' = -1$,

$$\begin{aligned} S_2 &= x^2\left(\frac{y}{x}\right)' \\ &= \frac{24(1-x^2)y^3 + 32x(1-x)y^2 - (9x^4+45x^3+209x^2+63x+18)y - 18x(x^3+x^2-x-1)}{27x^4+108x^3+418x^2+108x+27} \\ S_3 &= x^2\left(\frac{y^2}{x}\right)' \\ &= \frac{64x(1-x)y^3 + 9(x^4+2x^3-2x-1)y^2 + 12x(x^3+x^2-x-1)y + 48x^2(1-x^2)}{27x^4+108x^3+418x^2+108x+27} \end{aligned}$$

*and*

$$\begin{aligned} S_4 &= x^2\left(\frac{y^3}{x}\right)' \\ &= \frac{(27x^4+81x^3+209x^2+27x)y^3 + 18x(x^3+x^2-x-1)y^2 + 24x^2(x^2-1)y + 96x^3(1-x)}{27x^4+108x^3+418x^2+108x+27} \end{aligned}$$

so 2.8 becomes

$$M\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{2.10}$$

where

$$M = \begin{pmatrix} -1 & \frac{-18x(x^3+x^2-x-1)}{F} & \frac{48x^2(1-x^2)}{F} & \frac{96x^3(1-x)}{F} \\ 0 & \frac{-(9x^4+45x^3+209x^2+63x+18)}{F} & \frac{12x(x^3+x^2-x-1)}{F} & \frac{24x^2(x^2-1)}{F} \\ 0 & \frac{32x(1-x)}{F} & \frac{9(x^4+2x^3-2x-1)}{F} & \frac{18x(x^3+x^2-x-1)}{F} \\ 0 & \frac{24(1-x^2)}{F} & \frac{64x(1-x)}{F} & \frac{(27x^4+81x^3+209x^2+27x)}{F} \end{pmatrix}$$

and $F = 27x^4 + 108x^3 + 418x^2 + 108x + 27$. The system 2.10 admits a unique solution $f_1 = f_2 = 0, f_3 = -2$ and $f_4 = (x+1)/x$, whose denominator is not coprime with $V$, so the Hermite reduction is not applicable.

The above problem was first solved by Trager [Trag84], who proved that if $w$ is an *integral basis*, *i.e.* its elements generate $\mathbf{O}_{K[x]}$ over $K[x]$, then the system 2.8 always has a unique solution in $K(x)$ when $m > 1$, and that solution always has a denominator coprime with V. Furthermore, the denominator of each $w_i'$ must be squarefree, implying that the denominator of $h$ is a factor of $FUV^{m-1}$ where $F \in K[x]$ is squarefree and coprime with $UV$. He also described an algorithm for computing an integral basis, a necessary preprocessing for his Hermite reduction. The main problem with that approach is that computing the integral basis, whether by the method of [Trag84] or the local alternative [Hoei94], can be in general more expansive than the rest of the reduction process. We describe here the lazy Hermite reduction [Bron98], which avoids the precomputation of an integral basis. It is based on the observation that if $m > 1$ and 2.8 does not have a solution allowing us to perform the reduction, then either

- the $S_i$'s are linearly dependent over $K(x)$, or

- 2.8 has a unique solution in $K(x)$ whose denominator has a nontrivial common factor with $V$, or
- the denominator of some $w_i$ is not squarefree

In all of the above cases, we can replace our basis $w$ by a new one, also made up of integral elements, so that that $K[x]$-module generated by the new basis strictly contains the one generated by $w$:

**Theorem 1** ([Bron98]) *Suppose that $m \geq 2$ and that $\{S_1, \ldots, S_n\}$ as given by 2.9 are linearly dependent over $K(x)$, and let $T_1, \ldots, T_n \in K[x]$ be not all 0 and such that $\sum_{i=1}^n T_i S_i = 0$. Then,*

$$w_0 = \frac{U}{V} \sum_{i=1}^n T_i w_i \in \mathbf{O}_{K[x]}$$

*Furthermore, if $\gcd(T_1, \ldots, T_n) = 1$ then $w_0 \notin K[x]w_1 + \cdots + K[x]w_n$.*

**Theorem 2** ([Bron98]) *Suppose that $m \geq 2$ and that $\{S_1, \ldots, S_n\}$ as given by 2.9 are linearly independent over $K(x)$, and let $Q, T_1, \ldots, T_n \in K[x]$ be such that*

$$\sum_{i=1}^n A_i w_i = \frac{1}{Q} \sum_{i=1}^n T_i S_i$$

*Then,*

$$w_0 = \frac{U(V/\gcd(V,Q))}{\gcd(V,Q)} \sum_{i=1}^n T_i w_i \in \mathbf{O}_{K[x]}$$

*Furthermore, if $\gcd(Q, T_1, \ldots, T_n) = 1$ and $\deg(\gcd(V,Q)) \geq 1$, then $w_0 \notin K[x]w_1 + \cdots + K[x]w_n$.*

**Theorem 3** ([Bron98]) *Suppose that the denominator $F$ of some $w_i$ is not squarefree, and let $F = F_1 F_2^2 \cdots F_k^k$ be its squarefree factorization. Then,*

$$w_0 = F_1 \cdots F_k w_i' \in \mathbf{O}_{K[x]} \backslash (K[x]w_1 + \cdots + K[x]w_n).$$

The lazy Hermite reduction proceeds by solving the system 2.8 in $K(x)$. Either the reduction will succeed, or one of the above theorems produces an element $w_0 \in \mathbf{O}_{K[x]} \backslash (K[x]w_1 + \cdots + K[x]w_n)$. Let then $\sum_{i=1}^n C_i w_i$ and $F$ be the numerator and denominator of $w_0$ with respect to $w$. Using Hermitian row reduction, we can zero out the last row of

$$\begin{pmatrix} F & & & \\ & F & & \\ & & \ddots & \\ & & & F \\ C_1 & C_2 & \cdots & C_n \end{pmatrix}$$

obtaining a matrix of the form

$$\begin{pmatrix} C_{1,1} & C_{1,2} & \cdots & C_{1,n} \\ C_{2,1} & C_{2,2} & \cdots & C_{2,n} \\ \vdots & \vdots & & \vdots \\ C_{n,1} & C_{n,2} & \cdots & C_{n,n} \\ 0 & 0 & \cdots & 0 \end{pmatrix}$$

with $C_{ij} \in K[x]$. Let $\overline{w}_i = (\sum_{j=1}^{n} C_{ij} w_j)/F$ for $1 \leq i \leq n$. Then, $\overline{w} = (\overline{w}_1, \ldots, \overline{w}_n)$ is a basis for $E$ over $K$ and

$$K[x]\overline{w}_1 + \cdots + K[x]\overline{w}_n = K[x]w_1 + \cdots + K[x]w_n + K[x]w_0$$

is a submodule of $\mathbf{O}_{K[x]}$, which strictly contains $K[x]w_1 + \cdots + K[x]w_n$, since it contains $w_0$. Any strictly increasing chain of submodules of $\mathbf{O}_{K[x]}$ must stabilize after a finite number of steps, which means that this process produces a basis for which either the Hermite reduction can be carried out, or for which $f$ has a squarefree denominator.

**Example 2** Continuing example 1 for which the Hermite reduction failed, Theorem 2 implies that

$$w_0 = \frac{1}{x}(-2xw_3 + (x+1)w_4) = (-2xy^2 + (x+1)y^3)x \in \mathbf{O}_{K[x]}$$

Performing a Hermitian row reduction on

$$\begin{pmatrix} x & & & \\ & x & & \\ & & x & \\ & & & x \\ 0 & 0 & -2x & x+1 \end{pmatrix}$$

yields

$$\begin{pmatrix} x & & & \\ & x & & \\ & & x & \\ & & & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

so the new basis is $\overline{w} = (1, y, y^2, y^3/x)$, and the denominator of $f$ with respect to $\overline{w}$ is $x$, which is squarefree.

### 2.2.2 Simple radical extensions

The integration algorithm becomes easier when $E$ is a simple radical extension of $K(x)$, *i.e.* $E = K(x)[y]/(y^n - a)$ for some $a \in K(x)$. Write $a = A/D$ where $A, D \in K[x]$, and let $AD^{n-1} = A_1 A_2^2 \cdots A_k^k$ be a squarefree factorization of $AD^{n-1}$. Writing $i = nq_i + r_i$, for $1 \leq i \leq k$, where $0 \leq r_i < n$, let $F = A_1^{q_1} \cdots A_k^{q_k}$, $H = A_1^{r_1} \cdots A_k^{r_k}$ and $z = yD/F$. Then,

$$z^n = \left(y\frac{D}{F}\right)^n = \frac{y^n D^n}{F^n} = \frac{AD^{n-1}}{F} = A_1^{r_1} \cdots A_k^{r_k} = H$$

Since $r_i < n$ for each $i$, the squarefree factorization of $H$ is of the form $H = H_1 H_2^2 \cdots H_m^m$ with $m < n$. An integral basis is then $w = (w_1, \ldots, w_n)$ where

$$w_i = \frac{z^{i-1}}{\prod_{j=1}^{m} H_j^{\lfloor (i-1)j/n \rfloor}} \quad 1 \leq i \leq n \tag{2.11}$$

and the Hermite reduction with respect to the above basis is always guaranteed to succeed. Furthermore, when using that basis, the system 2.8 becomes diagonal and its solution can

be written explicitly: writing $D_i = \prod_{j=1}^{m} H_j^{\lfloor ij/n \rfloor}$ we have

$$\begin{aligned} S_i &= UV^m \left(\frac{w_i}{V^{m-1}}\right)' = UV^m \left(\frac{z^{i-1}}{D_{i-1}V^{m-1}}\right)' \\ &= UV^m \left(\frac{i-1}{n}\frac{H'}{H} - \frac{{D_{i-1}}'}{D_{i-1}} - (m-1)\frac{V'}{V}\right)\left(\frac{z^{i-1}}{D_{i-1}V^{m-1}}\right) \\ &= U\left(V\left(\frac{i-1}{n}\frac{H'}{H} - \frac{{D_{i-1}}'}{D_{i-1}}\right) - (m-1)V'\right) w_i \end{aligned}$$

so the unique solution of 2.8 in $K(x)$ is

$$f_i = \frac{A_i}{U\left(V\left(\frac{i-1}{n}\frac{H'}{H} - \frac{{D_{i-1}}'}{D_{i-1}}\right) - (m-1)V'\right)} \quad \text{for } 1 \le i \le n \tag{2.12}$$

and it can be shown that the denominator of each $f_i$ is coprime with $V$ when $m \ge 2$.

**Example 3** *Consider*

$$\int \frac{(2x^8+1)\sqrt{(x^8+1)}}{x^{17}+2x^9+x}\,dx$$

*The integrand is*

$$f = \frac{(2x^8+1)y}{x^{17}+2x^9+x} \in E = \mathbb{Q}(x)[y]/(y^2 - x^8 - 1)$$

*so $H = x^8+1$ which is squarefree, implying that the integral basis 2.11 is $(w_1, w_2) = (1, y)$. The squarefree factorization of $x^{17}+2x^9+x$ is $x(x^8+1)^2$ so $U = x$, $V = x^8+1$, $m = 2$, and the solution 2.12 of 2.8 is*

$$f_1 = 0, \quad f_2 = \frac{2x^8+1}{x\left((x^8+1)\frac{1}{2}\frac{8x^7}{x^8+1} - 8x^7\right)} = -\frac{(2x^8+1)/4}{x^8}$$

*We have $Q = x^8$, so $V - Q = 1$, $A = 1$, $R = -1$ and $RQf_2 = V/2 - 1/4$, implying that*

$$B = -\frac{y}{4} \quad \text{and} \quad h = f - \left(\frac{B}{V}\right)' = \frac{y}{x(x^8+1)}$$

*solve 2.7, i.e.*

$$\int \frac{(2x^8+1)\sqrt{(x^8+1)}}{x^{17}+2x^9+x}\,dx = -\frac{\sqrt{x^8+1}}{4(x^8+1)} + \int \frac{\sqrt{x^8+1}}{x(x^8+1)}\,dx$$

*and the remaining integrand has a squarefree denominator.*

### 2.2.3 Liouville's Theorem

Up to this point, the algorithms we have presented never fail, yet it can happen that an algebraic function does not have an elementary integral, for example

$$\int \frac{x\,dx}{\sqrt{1-x^3}}$$

which is not an elementary function of $x$. So we need a way to recognize such functions before completing the integration algorithm. Liouville was the first to state and prove a precise theorem from Laplace's observation that we can restrict the elementary integration problem by allowing only new logarithms to appear linearly in the integral, all the other terms appearing in the integral being already in the integrand.

**Theorem 4 (Liouville [Liou1833a, Liou1833b])** *Let $E$ be an algebraic extension of the rational function field $K(x)$, and $f \in E$. If $f$ has an elementary integral, then there exist $v \in E$, constants $c_1, \ldots, c_n \in \overline{K}$ and $u_1, \ldots, u_k \in E(c_1, \ldots, c_k)^*$ such that*

$$f = v' + c_1 \frac{u_1'}{u_1} + \cdots + c_k \frac{u_k'}{u_k} \tag{2.13}$$

The above is a restriction to algebraic functions of the strong Liouville Theorem, whose proof can be found in [Bron97, Risc69b]. An elegant and elementary algebraic proof of a slightly weaker version can be found in [Rose72]. As a consequence, we can look for an integral of the form 2.4, Liouville's Theorem guaranteeing that there is no elementary integral if we cannot find one in that form. Note that the above theorem does not say that every integral must have the above form, and in fact that form is not always the most convenient one, for example,

$$\int \frac{dx}{1+x^2} = arctan(x) = \frac{\sqrt{-1}}{2} \log\left(\frac{\sqrt{-1}+x}{\sqrt{-1}-x}\right)$$

### 2.2.4 The integral part

Following the Hermite reduction, we can assume that we have a basis $w = (w_1, \ldots, w_n)$ of $E$ over $K(x)$ made of integral elements such that our integrand is of the form $f = \sum_{i=1}^n A_i w_i / D$ where $D \in K[x]$ is squarefree. Given Liouville's Theorem, we now have to solve equation 2.13 for $v$, $u_1, \ldots, u_k$ and the constants $c_1, \ldots, c_k$. Since $D$ is squarefree, it can be shown that $v \in \mathbf{O}_{K[x]}$ for any solution, and in fact $v$ corresponds to the polynomial part of the integral of rational functions. It is however more difficult to compute than the integral of polynomials, so Trager [Trag84] gave a change of variable that guarantees that either $v' = 0$ or $f$ has no elementary integral. In order to describe it, we need to define the analogue for algebraic functions of having a nontrivial polynomial part: we say that $\alpha \in E$ is *integral at infinity* if there is a polynomial $p = \sum_{i=1}^m a_i y^i \in K[x][y]$ such that $p(x, \alpha) = 0$ and $\deg(a_m) \geq \deg(a_i)$ for each $i$. Note that a rational function $A/D \in K(x)$ is integral at infinity if and only if $\deg(A) \leq \deg(D)$ since it is a zero of $Dy - A$. When $\alpha - E$ is not integral at infinity, we say that it has a *pole at infinity*. Let

$$\mathbf{O}_\infty = \{\alpha \in E \text{ such that } \alpha \text{ is integral at infinity}\}$$

A set $(b_1, \ldots, b_n) \in E^n$ is called *normal at infinity* if there are $r_1, \ldots, r_n \in K(x)$ such that every $\alpha \in \mathbf{O}_\infty$ can be written as $\alpha = \sum_{i=1}^n B_i r_i b_i / C$ where $C, B_1, \ldots, B_n \in K[x]$ and $\deg(C) \geq \deg(B_i)$ for each $i$. We say that the differential $\alpha dx$ is integral at infinity if $\alpha x^{1+1/r} \in \mathbf{O}_\infty$ where $r$ is the smallest ramification index at infinity. Trager [Trag84] described an algorithm that converts an arbitrary integral basis $w_1, \ldots, w_n$ into one that is also normal at infinity, so the first part of his integration algorithm is as follows:

1. Pick any basis $b = (b_1, \ldots, b_n)$ of $E$ over $K(x)$ that is composed of integral elements.
2. Pick an integer $N \in \mathbb{Z}$ that is not zero of the denominator of $f$ with respect to $b$, nor of the discriminant of $E$ over $K(x)$, and perform the change of variable $x = N + 1/z$, $dx = -dz/z^2$ on the integrand.

3. Compute an integral basis $w$ for $E$ over $K(z)$ and make it normal at infinity
4. Perform the Hermite reduction on $f$ using $w$, this yields $g, h \in E$ such that $\int f\ dz = g + \int h\ dz$ and $h$ has a squarefree denominator with respect to $w$.
5. If $hz^2$ has a pole at infinity, then $\int f\ dz$ and $\int h\ dz$ are not elementary functions
6. Otherwise, $\int h\ dz$ is elementary if and only if there are constants $c_1, \ldots, c_k \in \overline{K}$ and $u_1, \ldots, u_k \in E(c_1, \ldots, c_k)^*$ such that

$$h = \frac{c_1}{u_1}\frac{du_1}{dz} + \cdots + \frac{c_k}{u_k}\frac{du_k}{dz} \tag{2.14}$$

The condition that $N$ is not a zero of the denominator of $f$ with respect to $b$ implies that the $fdz$ is integral at infinity after the change of variable, and Trager proved that if $hdz$ is not integral at infinity after the Hermite reduction, then $\int h\ dz$ and $\int f\ dz$ are not elementary functions. The condition that $N$ is not a zero of the discriminant of $E$ over $K(x)$ implies that the ramification indices at infinity are all equal to 1 after the change of variable, hence that $h\ dz$ is integral at infinity if and only if $hz^2 \in \mathbf{O}_\infty$. That second condition on $N$ can be disregarded, in which case we must replace $hz^2$ in step 5 by $hz^{1+1/r}$ where $r$ is the smallest ramification index at infinity. Note that $hz^2 \in \mathbf{O}_\infty$ implies that $hz^{1+1/r} \in \mathbf{O}_\infty$, but not conversely. Finally, we remark that for simple radical extensions, the integral basis 2.11 is already normal at infinity.

Alternatively, we can use lazy Hermite reduction in the above algorithm: in step 3, we pick any basis made of integral elements, then perform the lazy Hermite reduction in step 4. If $h \in K(z)$ after the Hermite reduction, then we can complete the integral without computing an integral basis. Otherwise, we compute an integral basis and make it normal at infinity between steps 4 and 5. This lazy variant can compute $\int f\ dx$ whenever it is an element of $E$ without computing an integral basis.

### 2.2.5 The logarithmic part

Following the previous sections, we are left with solving equation 2.14 for the constants $c_1, \ldots, c_k$ and for $u_1, \ldots, u_k$. We must make at this point the following additional assumptions:

- we have an integral primitive element for $E$ over $K(z)$, *i.e.* $y \in \mathbf{O}_{K[z]}$ such that $E = K(z)(y)$,
- $[E : K(z)] = [E : \overline{K}(z)]$, *i.e.* the minimal polynomial for $y$ over $K[z]$ is absolutely reducible, and
- we have an integral basis $w = (w_1, \ldots, w_n)$ for $E$ over $K(z)$, and $w$ is normal at infinity

A primitive element can be computed by considering linear combinations of the generators of $E$ over $K(x)$ with random coefficients in $K(x)$, and Trager [Trag84] describes an absolute factorization algorithm, so the above assumptions can be ensured, although those steps can be computationally very expensive, except in the case of simple radical extensions. Before describing the second part of Trager's integration algorithm, we need to define some concepts from the theory of algebraic curves. Given a finite algebraic extension $E = K(z)(y)$ of $K(z)$, a *place* $P$ of $E$ is a proper local subring of $E$ containing $K$, and a *divisor* is a formal sum $\sum n_P P$ with finite support, where the $n_P$'s are integers and the $P$'s are places. Let $P$ be a place, then its maximal ideal $\mu_P$ is principal, so let $p \in E$ be a generator of $\mu_P$. The *order at* $P$ is the function $\nu_P : E^* \to \mathbb{Z}$ which maps $f \in E^*$ to the largest $k \in \mathbb{Z}$ such that $f \in p^k P$. Given $f \in E^*$, the *divisor of* $f$ is $(f) = \sum \nu_P(f) P$ where the sum is taken over all the places.

It has finite support since $\nu_P(f) \neq 0$ if and only if $P$ is a pole or zero of $f$. Finally, we say that a divisor $\delta = \sum n_P P$ is *principal* if $\delta = (f)$ for some $f \in E^*$. Note that if $\delta$ is principal, the $\sum n_P = 0$, but the converse is not generally true, except if $E = K(z)$. Trager's algorithm proceeds essentially by constructing candidate divisors for the $u_i$'s of 2.14:

- Let $\sum_{i=1}^{n} A_i w_i$ be the numerator of $h$ with respect to $w$, and $D$ be its (squarefree) denominator
- Write $\sum_{i=1}^{n} A_i w_i = G/H$, where $G \in K[z, y]$ and $H \in K[z]$
- Let $f \in K[z, y]$ be the (monic) minimum polynomial for $y$ over $K(z)$, $t$ be a new indeterminante and compute

$$R(t) = \text{resultant}_{\text{z}} \left( \text{pp}_{\text{t}} \left( \text{resultant}_{\text{y}} \left( G - tH\frac{dD}{dz}, F \right) \right), D \right) \in K[t]$$

- Let $\alpha_1, \ldots, \alpha_s \in \overline{K}$ be the distinct nonzero roots of $R$, $(q_1, \ldots, q_k)$ be a basis for the vector space that they generate over $\mathbb{Q}$, write $\alpha_i = r_{i1}q_1 + \cdots + r_{ik}q_k$ for each $i$, where $r_{ij} \in \mathbb{Q}$ and let $m > 0$ be a common denominator for all the $r_{ij}$'s
- For $1 \leq j \leq k$, let $\delta_j = \sum_{i=1}^{s} m r_{ij} \sum_l r_l P_l$ where $r_l$ is the ramification index of $P_l$ and $P_l$ runs over all the places at which $h\ dz$ has residue $r_i \alpha_i$
- If there are nonzero integers $n_1, \ldots, n_k$ such that $n_j \delta_j$ is principal for each $j$, then let

$$u = h - \frac{1}{m} \sum_{j=1}^{k} \frac{q_j}{n_j u_j} \frac{du_j}{dz}$$

  where $u_j \in E(\alpha_1, \ldots, \alpha_s)^*$ is such that $n_j \delta_j = (u_j)$. If $u = 0$, then $\int h\ dz = \sum_{j=1}^{k} q_j \log(u_j)/(m n_j)$, otherwise if either $u \neq 0$ or there is no such integer $n_j$ for at least one $j$, then $h\ dz$ has no elementary integral.

Note that this algorithm expresses the integral, when it is elementary, with the smallest possible number of logarithms. Steps 3 to 6 requires computing in the splitting field $K_0$ of $R$ over $K$, but it can be proven that, as in the case of rational functions, $K_0$ is the minimal algebraic extension of $K$ necessary to express the integral in the form 2.4. Trager [Trag84] describes a representation of divisors as fractional ideals and gives algorithms for the arithmetic of divisors and for testing whether a given divisor is principal. In order to determine whether there exists an integer $N$ such that $N\delta$ is principal, we need to reduce the algebraic extension to one over a finite field $\mathbb{F}_{p^q}$ for some "good" prime $p \in \mathbb{Z}$. Over $\mathbb{F}_{p^q}$, it is known that for every divisor $\delta = \sum n_P P$ such that $\sum n_P = 0$, $M\delta$ is principal for some integer $1 \leq M \leq (1 + \sqrt{p^q})^{2g}$, where $g$ is the genus of the curve [Weil71], so we compute such an $M$ by testing $M = 1, 2, 3, \ldots$ until we find it. It can then be shown that for almost all primes $p$, if $M\delta$ is not principal in characteristic 0, the $N\delta$ is not principal for any integer $N \neq 0$. Since we can test whether the prime $p$ is "good" by testing whether the image in $\mathbb{F}_{p^q}$ of the discriminant of the discriminant of the minimal polynomial for $y$ over $K[z]$ is 0, this yields a complete algorithm. In the special case of hyperelliptic extensions, *i.e.* simple radical extensions of degree 2, Bertrand [Bert95] describes a simpler representation of divisors for which the arithmetic and principality tests are more efficient than the general methods.

**Example 4** *Continuing example 3, we were left with the integrand*

$$\frac{\sqrt{x^8 + 1}}{x(x^8 + 1)} = \frac{w_2}{x(x^8 + 1)} \in E = \mathbb{Q}(x)[y]/(y^2 - x^8 - 1)$$

*where $(w_1, w_2) = (1, y)$ is an integral basis normal at infinity, and the denominator $D = x(x^8 + 1)$ of the integrand is squarefree. Its numerator is $w_2 = y$, so the resultant of step 3 is*

$$resultant_x(pp_t(resultant_y(y - t(9x^8 + 1), y^2 - x^8 - 1)), x(x^8 + 1)) = ct^{16}(t^2 - 1)$$

*where $c$ is a large nonzero integer. Its nonzero roots are $\pm 1$, and the integrand has residue 1 at the place $P$ corresponding to the point $(x, y) = (0, 1)$ and $-1$ at the place $Q$ corresponding to the point $(x, y) = (0, -1)$, so the divisor $\delta_1$ of step 5 is $\delta_1 = P - Q$. It turns out that $\delta_1$, $2\delta_1$, and $3\delta_1$ are not principal, but that*

$$4\delta_1 = \left(\frac{x^4}{1+y}\right) \quad \text{and} \quad \frac{w_2}{x(x^8+1)} - \frac{1}{4}\frac{(x^4/(1+y))'}{x^4/(1+y)} = 0$$

*which implies that*

$$\int \frac{\sqrt{x^8+1}}{x(x^8+1)}\,dx = \frac{1}{4}\log\left(\frac{x^4}{1+\sqrt{x^8+1}}\right)$$

**Example 5** *Consider*

$$\int \frac{x\,dx}{\sqrt{1-x^3}}$$

*The integrand is*

$$f = \frac{xy}{1-x^3} \in E = \mathbb{Q}(x)[y]/(y^2 + x^3 - 1)$$

*where $(w_1, w_2) = (1, y)$ is an integral basis normal at infinity, and the denominaotr $D = 1 - x^3$ of the integrand is squarefree. Its numerator is $xw_2 = xy$, so the resultant of step 3 is*

$$resultant_x(pp_t(resultant_y(xy + 3tx^2, y^2 + x^3 - 1)), 1 - x^3) = 729t^6$$

*whose only root is 0. Since $f \neq 0$, we conclude from step 6 that $\int f\,dx$ is not an elementary function.*

**Example 6**

$$\int \frac{dx}{x\sqrt{1-x^3}}$$

*The integrand is*

$$f = \frac{y}{x-x^4} \in E = \mathbb{Q}(x)[y]/(y^2 + x^3 - 1)$$

*where $(w_1, w_2) = (1, y)$ is an integral basis normal at infinity, and the denominator $D = x - x^4$ of the integrand is squarefree. Its numerator is $w_2 = y$, so the resultant of step 3 is*

$$resultant_x(pp_t(resultant_y(y + t(4x^3 - 1), y^2 + x^3 - 1)), x - x^4) = 729t^6(t^2 - 1)$$

*Its nonzero roots are $\pm 1$, and the integrand has residue 1 at the place $P$ corrseponding to the point $(x, y) = (0, 1)$ and $-1$ at the place $Q$ corresponding to the point $(x, y) = (0, -1)$ so the divisor $\delta_1$ of step 5 is $\delta_1 = P - Q$. It turns out that $\delta_1$ and $2\delta_1$ are not principal, but that*

$$3\delta_1 = \left(\frac{y-1}{y+1}\right) \quad \text{and} \quad \frac{y}{x-x^4} - \frac{1}{3}\frac{((y-1)/(y+1))'}{(y-1)/(y+1)} = 0$$

*which implies that*

$$\int \frac{dx}{x\sqrt{1-x^3}} = \frac{1}{3}\log\left(\frac{\sqrt{1-x^3}-1}{\sqrt{1-x^3}+1}\right)$$

## 2.3 Elementary Functions

Let $f$ be an arbitrary elementary function. In order to generalize the algorithms of the previous sections, we need to build an algebraic model in which $f$ behaves in some sense like a rational or algebraic function. For that purpose, we need to formally define differential fields and elementary functions.

### 2.3.1 Differential algebra

A *differential field* $(K,')$ is a field $K$ with a given map $a \to a'$ from $K$ into $K$, satisfying $(a+b)' = a' + b'$ and $(ab)' = a'b + ab'$. Such a map is called a *derivation* on $K$. An element $a \in K$ which satisfies $a' = 0$ is called a *constant*, and the set $\mathrm{Const}(K)=\{a \in K$ such that $a' = 0\}$ of all the constants of $K$ is called a subfield of $K$.

A differential field $(E,')$ is a *differential equation* of $(K,')$ if $K \subseteq E$ and the derivation on $E$ extends the one on $K$. In that case, an element $t \in E$ is a *monomial* over $K$ if $t$ is transcendental over $K$ and $t' \in K[t]$, which implies that both $K[t]$ and $K(t)$ are closed under $'$. An element $t \in E$ is *elementary over* $K$ if either

- $t' = b'/b$ for some $b \in K^*$, in which case we say that $t$ is a *logarithm* over $K$, and write $t = log(b)$, or
- $t' = b't$ for some $b \in K^*$, in which case we say that $t$ is an *exponential* over $K$, and write $t = e^b$, or
- $t$ is algebraic over $K$

A differential extension $(E,')$ of $(K,')$ is *elementary over* $K$, if there exist $t_1, \ldots, t_m$ in $E$ such that $E = K(t_1, \ldots, t_m)$ and each $t_i$ is elementary over $K(t_1, \ldots, t_{i-1})$. We say that $f \in K$ *has an elementary integral* over $K$ if there exists an elementary extension $(F,')$ of $(K,')$ and $g \in F$ such that $g' = f$. An *elementary function* of the variable $x$ is an element of an elementary extension of the rational function field $(C(x), d/dx)$, where $C = \mathrm{Const}(C(x))$.

Elementary extensions are useful for modeling any function as a rational or algebraic function of one main variable over the other terms present in the function: given an elementary integrand $f(x)\ dx$, the integration algorithm first constructs a field $C$ containing all the constants appearing in $f$, then the rational function field $(C(x), d/dx)$, then an elementary tower $E = C(x)(t_1, \ldots, t_k)$ containing $f$. Note that such a tower is not unique, and in addition, ajoining a logarithm could in fact adjoin a new constant, and an exponential could in fact be algebraic, for example $\mathbb{Q}(x)(log(x), log(2x)) = \mathbb{Q}(log(2))(x)(log(x))$ and $\mathbb{Q}(x)(e^{log(x)/2}) = \mathbb{Q}(x)(\sqrt{x})$. There are however algorithms that detect all such occurences and modify the tower accordingly [Risc79], so we can assume that all the logarithms and exponentials appearing in $E$ are monomials, and that $\mathrm{Const}(E) = C$. Let now $k_0$ be the largest index such that $t_{k_0}$ is transcendental over $K = C(x)(t_1, \ldots, t_{k_0-1})$ and $t = t_{k_0}$. Then $E$ is a finitely generated algebraic extension of $K(t)$, and in the special case $k_0 = k$, $E = K(t)$. Thus, $f \in E$ can be seen as a univariate rational or algebraic function over $K$, the major difference with the pure rational or algebraic cases being that $K$ is not constant with respect to the derivation. It turns out that the algorithms of the previous section can be generalized to such towers, new methods being required only for the polynomial (or integral) part. We note that Liouville's Theorem remains valid when $E$ is an arbitrary differential field, so the integration algorithms work by attempting to solve equation 2.13 as previously.

**Example 7** *The function (1) is the element* $f = (t - t^{-1})\sqrt{-1}/2$ *of* $E = K(t)$ *where*

*$K = \mathbb{Q}(\sqrt{-1})(x)(t_1, t_2)$ with*

$$t_1 = \sqrt{x^3 - x + 1}, \quad t_2 = e^{2\sqrt{-1}(x^3 - t_1)}, \quad \text{and} \quad t = e^{((1-t_2)/(1+t_2)) - x\sqrt{-1}}$$

*which is transcendental over $K$. Alternatively, it can also be written as the element $f = 2\theta/(1+\theta^2)$ of $F = K(\theta)$ where $K == \mathbb{Q}(x)(\theta_1, \theta_2)$ with*

$$\theta_1 = \sqrt{x^3 - x + 1}, \quad \theta_2 = \tan(x^3 - \theta_1), \quad \text{and} \quad \theta = \tan\left(\frac{x + \theta_2}{2}\right)$$

*which is a transcendental monomial over $K$. It turns out that both towers can be used in order to integrate $f$.*

The algorithms of the previous sections relied extensively on squarefree factorization and on the concept of squarefree polynomials. The appropriate analogue in monomial extensions is the notion of *normal* polynomials: let $t$ be a monomial over $K$, we say that $p \in K[t]$ is *normal* (with respect to $'$) if $\gcd(p, p') = 1$, and that $p$ is *special* if $\gcd(p, p') = p$, *i.e.* $p | p'$ in $K[t]$. For $p \in K[t]$ squarefree, let $p_s = \gcd(p, p')$ and $p_n = p/p_s$. Then $p = p_s p_n$, while $p_s$ is special and $p_n$ is normal. Therefore, squarefree factorization can be used to write any $q \in K[t]$ as a product $q = q_s q_n$, where $\gcd(q_s, q_n) = 1$, $q_s$ is special and all the squarefree factors of $q_n$ are normal. We call $q_s$ the *special part* of $q$ and $q_n$ its *normal part*.

### 2.3.2 The Hermite reduction

The Hermite reductions we presented for rational and algebraic functions work in exactly the same way algebraic extensions of monomial extensions of $K$, as long as we apply them only to the normal part of the denominator of the integrand. Thus, if $D$ is the denominator of the integrand, we let $S$ be the special part of $D$, $D_1 D_2^2 \dots D_m^m$ be a squarefree factorization of the *normal* part of $D$, $V = D_m$, $U = D/V^m$ and the rational and algebraic Hermite reductions proceed normally, eventually yielding an integrand whose denominator has a squarefree normal part.

**Example 8** *Consider*

$$\int \frac{x - \tan(x)}{\tan(x)^2}\, dx$$

*The integrand is*

$$f = \frac{x - t}{t^2} \in K(t) \quad \text{where } K = \mathbb{Q}(x) \text{ and } t' = t^2 + 1$$

*Its denominator is $D = t^2$, and $\gcd(t, t') = 1$ implying that $t$ is normal, so $m = 2$, $V = t$, $U = D/t^2 = 1$, and the extended Euclidean algorithm yields*

$$\frac{A}{1 - m} = t - x = -x(t^2 + 1) + (xt + 1)t = -xUV' + (xt + 1)V$$

*implying that*

$$\int \frac{x - \tan(x)}{\tan(x)^2}\, dx = -\frac{x}{\tan(x)} - \int x\, dx$$

*and the remaining integrand has a squarefree denominator.*

**Example 9** *Consider*

$$\int \frac{\log(x)^2 + 2x\log(x) + x^2 + (x+1)\sqrt{x + \log(x)}}{x\log(x)^2 + 2x^2\log(x) + x^3}\, dx$$

*The integrand is*

$$f = \frac{t^2 + 2xt + x^2 + (x+1)y}{xt^2 + 2x^2t + x^3} \in E = K(t)[y]/(y^2 - x - t)$$

*where* $K = \mathbb{Q}(x)$ *and* $t = log(x)$. *The denominator of* $f$ *with respect to the basis* $w = (1, y)$ *is* $D = xt^2 + 2x^2t + x^3$ *whose squarefree factorization is* $x(t+x)^2$. *Both* $x$ *and* $t + x$ *are normal, so* $m = 2$, $V = t + x$, $U = D/V^2 = x$, *and the solution 2.12 of 2.8 is*

$$f_1 = \frac{t^2 + 2xt + x^2}{x(-(t'+1))} = -\frac{t^2 + 2xt + x^2}{x+1},$$

$$f_2 = \frac{x+1}{x\left((t+x)\frac{1}{2}\frac{t'+1}{t+z} - (t'+1)\right)} = -2$$

*We have* $Q = 1$, *so* $0V + 1Q = 1$, $A = 0$, $R = 1$, $RQf_1 = f_1 = -V^2/(x+1)$ *and* $RQf_2 = f_2 = 0V - 2$, *so* $B = -2y$ *and*

$$h = f - \left(\frac{B}{V}\right)' = \frac{1}{x}$$

*implying that*

$$\int \frac{\log(x)^2 + 2x\log(x) + x^2 + (x+1)\sqrt{x + \log(x)}}{x\log(x)^2 + 2x^2\log(x) + x^2}\, dx = \frac{2}{\sqrt{x + log(x)}} + \int \frac{dx}{x}$$

*and the remaining integrand has a squarefree denominator.*

### 2.3.3 The polynomial reduction

In the transcendental case $E = K(t)$ and when $t$ is a monomial satisfying $\deg_t(t') \geq 2$, then it is possible to reduce the degree of the polynomial part of the integrand until it is smaller than $\deg_t(t')$. In the case when $t = \tan(b)$ for some $b \in K$, then it is possible either to prove that the integral is not elementary, or to reduce the polynomial part of the integrand to be in $K$. Let $f \in K(t)$ be our integrand and write $f = P + A/D$, where $P, A, D \in K[t]$ and $\deg(A) < \deg(D)$. Write $P = \sum_{i=1}^{e} p_i t^i$ and $t' = \sum_{i=0}^{d} c_i t^i$ where $p_0, \ldots, p_e, c_0, \ldots, c_d \in K$, $d \geq 2$, $p_e \neq 0$ and $c_d \neq 0$. It is easy to verify that if $e \geq d$, then

$$P = \left(\frac{a_e}{(e-d+1)c_d} t^{e-d+1}\right)' + \overline{P} \tag{2.15}$$

where $\overline{P} \in K[t]$ is such that $\overline{P} = 0$ or $\deg_t(\overline{P}) < e$. Repeating the above transformation we obtain $Q, R \in K[t]$ such that $R = 0$ or $\deg_t(R) < d$ and $P = Q' + R$. Write then $R = \sum_{i=0}^{d-1} r_i t^i$ where $r_0, \ldots, r_{d-1} \in K$. Again, it is easy to verify that for any *special* $S \in K[t]$ with $\deg_t(S) > 0$, we have

$$R = \frac{1}{\deg_t(S)} \frac{r_{d-1}}{c_d} \frac{S'}{S} + \overline{R}$$

where $\overline{R} \in K[t]$ is such that $\overline{R} = 0$ or $\deg_t(\overline{R}) < e - 1$. Furthermore, it can be proven [Bron97] that if $R + A/D$ has an elementary integral over $K(t)$, then $r_{d-1}/c_d$ is a constant, which implies that

$$\int R = \frac{1}{\deg_t(S)} \frac{r_{d-1}}{c_d} \log(S) + \int \left(\overline{R} + \frac{A}{D}\right)$$

so we are left with an integrand whose polynomial part has degree at most $\deg_t(t') - 2$. In this case $t = \tan(b)$ for $b \in K$, then $t' = b't^2 + b'$, so $\overline{R} \in K$.

**Example 10** *Consider*

$$\int (1 + x\tan(x) + \tan(x)^2)\ dx$$

*The integrand is*

$$f = 1 + xt + t^2 \in K(t) \quad \text{where } K = \mathbb{Q}(x) \text{ and } t' = t^2 + 1$$

*Using 2.15, we get* $\overline{P} = f - t' = f - (t^2 + 1) = xt$ *so*

$$\int (1 + x\tan(x) + \tan(x)^2)\ dx = \tan(x) + \int x\tan(x)\ dx$$

*and since* $x' \neq 0$*, the above criterion imples that the remaining integral is not an elementary function.*

### 2.3.4 The residue criterion

Similarly to the Hermite reduction, the Rothstein-Trager and Lazard-Rioboo-Trager algorithms are easy to generalize to the transcendental case $E = K(t)$ for arbitrary monomials $t$: let $f \in K(t)$ be our integrand and write $f = P + A/D + B/S$ where $P, A, D, B, S \in K[t]$, $\deg(A) < \deg(D)$, $S$ is special and, following the Hermite reduction, $D$ is normal. Let then $z$ be a new indeterminate, $\kappa : K[z] \to K[z]$ be give by $\kappa(\sum_i a_i z^i) = \sum_i a_i' z^i$,

$$R = \text{resultant}_t(D, A - zD') \in K[z]$$

be the Rothstein-Trager resultant, $R = R_1 R_2^2 \dots R_k^k$ be its squarefree factorization, $Q_i = \gcd_z(R_i, \kappa(R_i))$ for each $i$, and

$$g = \sum_{i=1}^{k} \sum_{a | Q_i(a) = 0} a \log(\gcd_t(D, A - aD'))$$

Note that the roots of each $Q_i$ must all be constants, and that the arguments of the logarithms can be obtained directly from the subresultant PRS of $D$ and $A - zD'$ as in the rational function case. It can then be proven [Bron97] that

- $f - g'$ is always "simpler" than $f$
- the splitting field of $Q_1 \cdots Q_k$ over $K$ is the minimal algebraic extension of $K$ needed in order to express $\int f$ in the form 2.4
- if $f$ has an elementary integral over $K(t)$, then $R | \kappa(R)$ in $K[z]$ and the denominator of $f - q'$ is special

Thus, while in the pure rational function case the remaining integrand is a polynomial, in this case the remaining integrand has a special denominator. In that case we have additionally that if its integral is elementary, then 2.13 has a solution such that $v \in K(t)$ has a special denominator, and each $u_i \in K(c_1, \dots, c_k)[t]$ is special.

**Example 11** *Consider*

$$\int \frac{2\log(x)^2 - \log(x) - x^2}{\log(x)^3 - x^2\log(x)}\ dx$$

*The integrand is*

$$f = \frac{2t^2 - t - x^2}{t^2 - xt^2} \in K(t) \quad \text{where } K = \mathbb{Q}(x) \text{ and } t = \log(x)$$

*Its denominator is* $D = t^3 - x^2 t$, *which is normal, and the resultant is*

$$\begin{aligned} R &= resultant_t\left(t^3 - x^2 t, \frac{2x - 3z}{x} t^2 + (2xz - 1)t + x(z - x)\right) \\ &= 4x^3(1 - x^2)\left(z^3 - xz^2 - \frac{1}{4}z + \frac{x}{4}\right) \end{aligned}$$

*which is squarefree in* $K[z]$. *We have*

$$\kappa(R) = -x^2(4(5x^2 + 3)z^3 + 8x(3x^2 - 2)z^2 + (5x^2 - 3)z - 2x(3x^2 - 2))$$

*so*

$$Q_1 = \gcd_z(R, \kappa R) = x^2\left(z^2 - \frac{1}{4}\right)$$

*and*

$$\gcd_t\left(t^3 + x^2 t, \frac{2x - 3a}{x} t^2 + (2xa - 1)t + x(a - x)\right) = t + 2ax$$

*where* $a^2 - 1/4 = 0$, *whence*

$$g = \sum_{a | a^2 - 1/4 = 0} a \log(t + 2ax) = \frac{1}{2}\log(t + x) - \frac{1}{2}\log(t - x)$$

*Computing* $f - g'$ *we find*

$$\int \frac{2\log(x)^2 - \log(x) - x^2}{\log(x)^3 - x^2\log(x)}\, dx = \frac{1}{2}\log\left(\frac{\log(x) + x}{\log(x) - x}\right) + \int \frac{dx}{\log(x)}$$

*and since* $\deg_z(Q_1) < \deg_z(R)$, *it follows that the remaining integral is not an elementary function (it is in fact the logarithmic integral* $Li(x)$*).*

In the most general case, when $E = K(t)(y)$ is algebraic over $K(t)$ and $y$ is integral over $K[t]$, the criterion part of the above result remains valid: let $w = (w_1, \ldots, w_n)$ be an integral basis for $E$ over $K(t)$ and write the integrand $f \in E$ as $f = \sum_{i=1}^n A_i w_i / D + \sum_{i=1}^n B_i w_i / S$ where $S$ is special and, following the Hermite reduction, $D$ is normal. Write $\sum_{i=1}^n A_i w_i = G/H$, where $G \in K[t, y]$ and $H \in K[t]$, let $F \in K[t, y]$ be the (monic) minimum polynomial for $y$ over $K(t)$, $z$ be a new indeterminante and compute

$$R(z) = \text{resultant}_\text{t}(\text{pp}_\text{z}(\text{resultant}_\text{y}(G - tHD', F)), D) \in K[t] \tag{2.16}$$

It can then be proven [Bron90c] that if $f$ has an elementary integral over $E$, then $R | \kappa(R)$ in $K[z]$.

**Example 12** *Consider*

$$\int \frac{\log(1 + e^x)^{(1/3)}}{1 + \log(1 + e^x)}\, dx \tag{2.17}$$

*The integrand is*

$$f = \frac{y}{t + 1} \in E = K(t)[y]/(y^3 - t)$$

*where* $K = \mathbb{Q}(x)(t_1)$, $t_1 = e^x$ *and* $t = \log(1 + t_1)$. *Its denominator with respect to the integral basis* $w = (1, y, y^2)$ *is* $D = t + 1$, *which is normal, and the resultant is*

$$R = \text{resultant}_\text{t}(\text{pp}_\text{z}(\text{resultant}_\text{y}(y - zt_1/(1 + t_1), y^3 - t)), t + 1) = -\frac{t_1^3}{(1 + t_1)^3}z^3 - 1$$

*We have*

$$\kappa(R) = -\frac{3t_1^3}{(1 + t_1)^4}z^3$$

*which is coprime with* $R$ *in* $K[z]$, *implying that the integral 2.17 is not an elementary function.*

### 2.3.5 The transcendental logarithmic case

Suppose now that $t = \log(b)$ for some $b \in K^*$, and that $E = K(t)$. Then, every special polynomial must be in $K$, so, following the residue criterion, we must look for a solution $v \in K[t]$, $u_1, \ldots, u_k \in K(c_1, \ldots, c_n)^*$ of 2.13. Furthermore, the integrand $f$ is also in $K[t]$, so write $f = \sum_{i=0}^{d} f_i t^i$ where $f_0, \ldots, f_d \in K$ and $f_d \neq 0$. We must have $\deg_t(v) \leq d + 1$, so writing $v = \sum_{i=0}^{d+1} v_i t^i$, we get

$$\int f_d t^d + \cdots + f_1 t + f_0 = v_{d+1}t^{d+1} + \cdots + v_1 t + v_0 + \sum_{i=1}^{k} c_i \log(u_i)$$

If $d = 0$, then the above is simply an integration problem for $f_0 \in K$, which can be solved recursively. Otherwise, differentiating both sides and equating the coefficients of $t^d$, we get ${v_{d+1}}' = 0$ and

$$f_d = v_d' + (d + 1)v_{d+1}\frac{b'}{b} \tag{2.18}$$

Since $f_d \in K$, we can recursively apply the integration algorithm to $f_d$, either proving that 2.18 has no solution, in which case $f$ has no elementary integral, or obtaining the constant $v_{d+1}$, and $v_d$ up to an additive constant (in fact, we apply recursively a specialized version of the integration algorithm to equations of the form 2.18, see [Bron97] for details). Write then $v_d = \overline{v_d} + c_d$ where $\overline{v_d} \in K$ is known and $c_d \in \text{Const}(K)$ is undetermined. Equating the coefficients of $t^{d-1}$ yields

$$f_{d-1} - d\overline{v_d}\frac{b'}{b} = {v_{d-1}}' + dc_d\frac{b'}{b}$$

which is an equation of the form 2.18, so we again recursively compute $c_d$ and $v_{d-1}$ up to an additive constant. We repeat this process until either one of the recursive integrations fails, in which case $f$ has no elementary integral, or we reduce our integrand to an element of $K$, which is then integrated recursively. The algorithm of this section can also be applied to real arc-tangent extensions, *i.e.* $K(t)$ where $t$ is a monomial satisfying $t' = b'/(1 + b^2)$ for some $b \in K$.

### 2.3.6 The transcendental exponential case

Suppose now that $t = e^b$ for some $b \in K$, and that $E = K(t)$. Then, every nonzero special polynomial must be of the form $at^m$ for $a \in K^*$ and $m \in \mathbb{N}$. Since

$$\frac{(at^m)'}{at^m} = \frac{a'}{a} + m\frac{t'}{t} = \frac{a'}{a} + mb'$$

we must then look for a solution $v \in K[t,t^{-1}]$, $u_1,\ldots,u_k \in K(c_1,\ldots,c_n)^*$ of 2.13. Furthermore, the integrand $f$ is also in $K[t,t^{-1}]$, so write $f = \sum_{i=e}^{d} f_i t^i$ where $f_e,\ldots,f_d \in K$ and $e,d \in \mathbb{Z}$. Since $(at^m)' = (a' + mb')t^m$ for any $m \in \mathbb{Z}$, we must have $v = Mb + \sum_{i=e}^{d} v_i t^i$ for some integer $M$, hence

$$\int \sum_{i=e}^{d} f_i t^i = Mb + \sum_{i=e}^{d} v_i t^i + \sum_{i=1}^{k} c_i \log(u_i)$$

Differentiating both sides and equating the coefficients of each power to $t^d$, we get

$$f_0 = (v_0 + Mb)' + \sum_{i=1}^{k} c_i \frac{u_i'}{u_i}$$

which is simply an integration problem for $f_0 \in K$, and

$$f_i = v_i' + ib'v_i \quad \text{for } e \le i \le d, i \ne 0$$

The above problem is called a *Risch differential equation over K*. Although solving it seems more complicated than solving $g' = f$, it is actually simpler than an integration problem because we look for the solutions $v_i$ in $K$ only rather than in an extension of $K$. Bronstein [Bron90c, Bron91a, Bron97] and Risch [Risc68, Risc69a, Risc69b] describe algorithms for solving this type of equation when $K$ is an elementary extension of the rational function field.

### 2.3.7 The transcendental tangent case

Suppose now that $t = \tan(b)$ for some $b \in K$, *i.e.* $t' = b'(1+t^2)$, that $\sqrt{-1} \notin K$ and that $E = K(t)$. Then, every nonzero special polynomial must be of the form $a(t^2+1)^m$ for $a \in K^*$ and $m \in \mathbb{N}$. Since

$$\frac{(a(t^2+1)^m)'}{a(t^2+1)^m} = \frac{a'}{a} + m\frac{(t^2+1)'}{t^2+1} = \frac{a'}{a} + 2mb't$$

we must look for $v = V/(t^2+1)^m$ where $V \in K[t]$, $m_1,\ldots,m_k \in \mathbb{N}$, constants $c_1,\ldots,c_k \in \overline{K}$ and $u_1,\ldots,u_k \in K(c_1,\ldots,c_k)^*$ such that

$$f = v' + 2b't\sum_{i=1}^{k} c_i m_i + \sum_{i=1}^{k} c_i \frac{u_i'}{u_i}$$

Furthermore, the integrand $f \in K(t)$ following the residue criterion must be of the form $f = A/(t^2+1)^M$ where $A \in K[t]$ and $M \ge 0$. If $M > 0$, it can be shown that $m = M$ and that

$$\begin{pmatrix} c' \\ d' \end{pmatrix} + \begin{pmatrix} 0 & -2mb' \\ 2mb' & 0 \end{pmatrix} \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} a \\ b \end{pmatrix} \tag{2.19}$$

where $at+b$ and $ct+d$ are the remainders module $t^2+1$ of $A$ and $V$ respectively. The above is a coupled differential system, which can be solved by methods similar to the ones used for Risch differential equations [Bron97]. If it has no solution, then the integral is not elementary, otherwise we reduce the integrand to $h \in K[t]$, at which point the polynomial

reduction either proves that its integral is not elementary, or reduce the integrand to an element of $K$, which is integrated recursively.

**Example 13** *Consider*

$$\int \frac{sin(x)}{x}\, dx$$

*The integrand is*

$$f = \frac{2t/x}{t^2+1} \in K(t) \quad \text{where } K = \mathbb{Q}(x) \text{ and } t = \tan\left(\frac{x}{2}\right)$$

*Its denominator is $D = t^2 + 1$, which is special, and the system 2.19 becomes*

$$\begin{pmatrix} c' \\ d' \end{pmatrix} + \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} 2/x \\ 0 \end{pmatrix}$$

*which has no solution in $\mathbb{Q}(x)$, implying that the integral is not an elementary function.*

### 2.3.8 The algebraic logarithmic case

The transcendental logarithmic case method also generalizes to the case when $E = K(t)(y)$ is algebraic over $K(t)$, $t = log(b)$ for $b \in K^*$ and $y$ is integral over $K[t]$: following the residue criterion, we can assume that $R|\kappa(R)$ where $R$ is given by 2.16, hence that all its roots in $\overline{K}$ are constants. The polynomial part of the integrand is replace by a family of at most $[E : K(t)]$ Puiseux expansions at infinity, each of the form

$$a_{-m}\theta^{-m} + \cdots + a_{-1}\theta^{-1} + \sum_{i\geq 0} a_i\theta^i \tag{2.20}$$

where $\theta^r = t^{-1}$ for some positive integer $r$. Applying the integration algorithm recursively to $a_r \in \overline{K}$, we can test whether there exist $\rho \in \text{Const}(\overline{K})$ and $v \in \overline{K}$ such that

$$a_r = v' + \rho\frac{b'}{b}$$

If there are no such $v$ and $c$ for at least one of the series, then the integral is not elementary, otherwise $\rho$ is uniquely determined by $a_r$, so let $\rho_1, \ldots, \rho_q$ where $q \leq [E : K(t)]$ be the distinct constants we obtain, $\alpha_1, \ldots, \alpha_s \in \overline{K}$ be the distinct nonzero roots of $R$, and $(q_1, \ldots, q_k)$ be a basis for the vector space generated by the $\rho_i$'s and $\alpha_i$'s over $\mathbb{Q}$. Write $\alpha_i = r_{i1}q_1 + \cdots + r_{ik}q_k$ and $\rho_i = s_{i1}q_1 + \cdots + s_{ik}q_k$ for each $i$, where $r_{ij}, s_{ij} \in \mathbb{Q}$ and let $m > 0$ be a common denominator for all the $r_{ij}$'s and $s_{ij}$'s. For $1 \leq j \leq k$, let

$$\delta_j = \sum_{i=1}^{s} m r_{ij} \sum_l r_l P_l - \sum_{i=1}^{q} m s_{ij} \sum_l s_l Q_l$$

where $r_l$ is the ramification index of $P_l$, $s_l$ is the ramification index of $Q_l$, $P_l$ runs over all the finite places at which $h\ dz$ has residue $r_l\alpha_i$ and $Q_l$ runs over all the infinite places at which $\rho = \rho_i$. As in the pure algebraic case, if there is a $j$ for which $N\delta_j$ is not principal for any nonzero integer $N$, then the integral is not elementary, otherwise, let $n_1, \ldots, n_k$ be nonzero integers such that $n_j\delta_j$ is principal for each $j$, and

$$h = f - \frac{1}{m}\sum_{j=1}^{k} \frac{q_j}{n_j}\frac{u_j'}{u_j}$$

where $f$ is the integrand and $u_j \in E(\alpha_1, \ldots, \alpha_s, \rho_1, \ldots, \rho_q)^*$ is such that $n_j\delta_j = (u_j)$. If the integral of $h$ is elementary, then 2.13 must have a solution with $v \in \mathbf{O}_{K[x]}$ and $u_1, \ldots, u_k \in \overline{K}$ so we must solve

$$h = \frac{\sum_{i=1}^n A_i w_i}{D} = \sum_{i=1}^n v_i' w_i + \sum_{i=1}^n v_i w_i' + \sum_{i=1}^k c_i \frac{u_i'}{u_i} \tag{2.21}$$

for $v_1, \ldots, v_n \in K[t]$, constants $c_1, \ldots, c_n \in \overline{K}$ and $u_1, \ldots, u_k \in \overline{K}^*$ where $w = (w_1, \ldots, w_n)$ is an integral basis for $E$ over $K(t)$.

If $E$ is a simple radical extension of $K(t)$, and we use the basis 2.11 and the notation of that section, then $w_1 = 1$ and

$$w_i' = \left(\frac{i-1}{n}\frac{H'}{H} - \frac{D_{i-1}'}{D_{i-1}}\right) w_i \quad \text{for } 1 \le i \le n \tag{2.22}$$

This implies that 2.21 becomes

$$\frac{A_1}{D} = v_1' + \sum_{i=1}^k c_i \frac{u_i'}{u_i} \tag{2.23}$$

which is simply an integration problem for $A_1/D \in K(t)$, and

$$\frac{A_i}{D} = v_i' + \left(\frac{i-1}{n}\frac{H'}{H} - \frac{D_{i-1}'}{D_{i-1}}\right) v_i \quad \text{for } 1 < i \le n \tag{2.24}$$

which are Risch differential equations over $K(t)$

**Example 14** *Consider*

$$\int \frac{(x^2+2x+1)\sqrt{x+\log(x)} + (3x+1)\log(x) + 3x^2 + x}{(x\log(x)+x^2)\sqrt{x+\log(x)} + x^2\log(x) + x^3}\, dx$$

*The integrand is*

$$f = \frac{((3x+1)t - x^3 + x^2)y - (2x^2 - x - 1)t - 2x^3 + x^2 + x}{xt^2 - (x^3 - 2x^2)t - x^4 + x^3} \in E = K(t)[y]/(F)$$

*where* $F = y^2 - x - t$, $K = \mathbb{Q}(x)$ *and* $t = \log(x)$. *Its denominator with respect to the integral basis* $w = (1, y)$ *is* $D = xt^2 - (x^3 - 2x^2)t - x^4 + x^3$, *which is normal, and the resultant is*

$$\begin{aligned} R &= \text{resultant}_{\text{t}}(\text{pp}_{\text{z}}(\text{resultant}_{\text{y}}(((3x+1)t - x^3 + x^2)y \\ &\qquad - (2x^2 - x - 1)t - 2x^3 + x^2 + x - zD', F)), D) \\ &= x^{12}(2x+1)^2(x+1)^2(x-1)^2 z^3(z-2) \end{aligned}$$

*We have*

$$\kappa(R) = \frac{36x^3 + 16x^2 - 28x - 12}{x(2x+1)(x+1)(x-1)} R$$

*so* $R|\kappa(R)$ *in* $K[z]$. *Its only nonzero root is 2, and the integrand has residue 2 at the place* $P$ *corresponding to the point* $(t, y) = (x^2 - x, -x)$. *There is only one place* $Q$ *at infinity of ramification index 2, and the coefficient of* $t^{-1}$ *in the Puiseux expansion of* $f$ *at* $Q$ *is*

$$a_2 = 1 - 2x + \frac{1}{x} = (x - x^2)' + \frac{x'}{x}$$

*which implies that the corresponding $\rho$ is 1. Therefore, the divisor for the logand is $\delta = 2P - 2Q$. It turns out that $\delta = (u)$ where $u = (x+y)^2 \in E^*$, so the new integrand is*

$$h = f - \frac{u'}{u} = f - 2\frac{(x+y)'}{x+y} = \frac{(x+1)y}{xt+x^2}$$

*We have $y^2 = t + x$, which is squarefree, so 2.23 becomes*

$$0 = v_1' + \sum_{i=1}^{k} c_i \frac{u_i'}{u_i}$$

*whose solution is $v_1 = k = 0$ and 2.24 becomes*

$$\frac{x+1}{xt+x^2} = v_2' + \frac{x+1}{2xt+2x^2} v_2$$

*whose solution is $v_2 = 2$, implying that $h = 2y'$, hence that*

$$\int \frac{(x^2+2x+1)\sqrt{x+\log(x)} + (3x+1)\log(x) + 3x^2 + x}{(x\log(x)+x^2)\sqrt{x+\log(x)} + x^2\log(x) + x^3}\, dx =$$

$$2\sqrt{x+\log(x)} + 2\log\left(x + \sqrt{x+\log(x)}\right)$$

In the general case when $E$ is not a radical extension of $K(t)$, 2.21 is solved by bounding $\deg_t(v_i)$ and comparing the Puiseux expansions at infinity of $\sum_{i=1}^{n} v_i w_i$ with those of the form 2.20 of $h$, see [Bron90c, Risc68] for details.

### 2.3.9 The algebraic exponential case

The transcendental exponential case method also generalizes to the case when $E = K(t)(y)$ is algebraic over $K(t)$, $t = e^b$ for $b \in K$ and $y$ is integral over $K[t]$: following the residue criterion, we can assume that $R|\kappa(R)$ where $R$ is given by 2.16, hence that all its roots in $\overline{K}$ are constants. The denominator of the integrand must be of the form $D = t^m U$ where $\gcd(U,t) = 1$, $U$ is squarefree and $m \geq 0$.

If $m > 0$, $E$ is a simple radical extension of $K(t)$, and we use the basis 2.11, then it is possible to reduce the power of $t$ appearing in $D$ by a process similar to the Hermite reduction: writing the integrand $f = \sum_{i=1}^{n} A_i w_i/(t^m U)$, we ask whether we can compute $b_1, \ldots, b_n \in K$ and $C_1, \ldots, C_n \in K[t]$ such that

$$\int \frac{\sum_{i=1}^{n} A_i w_i}{t^m U} = \frac{\sum_{i=1}^{n} b_i w_i}{t^m} + \int \frac{\sum_{i=1}^{n} C_i w_i}{t^{m-1} U}$$

Differentiating both sides and multiplying through by $t^m$ we get

$$\frac{\sum_{i=1}^{n} A_i w_i}{U} = \sum_{i=1}^{n} b_i' w_i + \sum_{i=1}^{n} b_i w_i' - m b' \sum_{i=1}^{n} b_i w_i + \frac{t\sum_{i=1}^{n} C_i w_i}{U}$$

Using 2.22 and equating the coefficients of $w_i$ on both sides, we get

$$\frac{A_i}{U} = b_i' + (\omega_i - m b') b_i + \frac{t C_i}{U} \quad \text{for } 1 \leq i \leq n \tag{2.25}$$

where

$$\omega_i = \frac{i-1}{n}\frac{H'}{H} - \frac{D'_{i-1}}{D_{i-1}} \in K(t)$$

Since $t'/t = b' \in K$, it follows that the denominator of $\omega_i$ is not divisible by $t$ in $K[t]$, hence, evaluating 2.25 at $t = 0$, we get

$$\frac{A_i(0)}{U(0)} = b'_i + (\omega_i(0) - mb')b_i \quad \text{for } 1 \le i \le n \tag{2.26}$$

which are Risch differential equations over $K(t)$. If any of them has no solution in $K(t)$, then the integral is not elementary, otherwise we repeat this process until the denominator of the integrand is normal. We then perform the change of variable $\overline{t} = t^{-1}$, which is also exponential over $K$ since $\overline{t}' = -b'\overline{t}$, and repeat the above process in order to eliminate the power of $\overline{t}$ from the denominator of the integrand. It can be shown that after this process, any solution of 2.13 must have $v \in K$.

**Example 15** *Consider*

$$\int \frac{3(x+e^x)^{(1/3)} + (2x^2+3x)e^x + 5x^2}{x(x+e^x)^{(1/3)}}\,dx$$

*The integrand is*

$$f = \frac{((2x^2+3x)t + 5x^2)y^2 + 3t + 3x}{xt + x^2} \in E = K(t)[y]/(y^3 - t - x)$$

*where* $K = \mathbb{Q}(x)$ *and* $t = e^x$. *Its denominator with respect to the integral basis* $w = (1, y, y^2)$ *is* $D = xt + x^2$, *which is normal, and the resultant is*

$$R = \mathrm{resultant_t}(\mathrm{pp_z}(\mathrm{resultant_y}(((2x^2+3x)t + 5x^2)y^2 + 3t + 3x - zD', y^3 - t - x)), D) = x^8(1-x)^3z^3$$

*We have*

$$\kappa(R) = \frac{11x - 8}{x(x-1)}R$$

*so* $R|\kappa(R)$ *in* $K[z]$, *its only root being 0. Since* $D$ *is not divisible by* $t$, *let* $\overline{t} = t^{-1}$ *and* $z = \overline{t}y$. *We have* $\overline{t}' = -\overline{t}$ *and* $z^3 - \overline{t}^2 - x\overline{t}^3 = 0$, *so the integral basis 2.11 is*

$$\overline{w} = (\overline{w}_1, \overline{w}_2, \overline{w}_3) = \left(1, z, \frac{z^2}{\overline{t}}\right)$$

*Writing* $f$ *in terms of that basis gives*

$$f = \frac{3x\overline{t}^2 + 3\overline{t} + (5x^2\overline{t} + 2x^2 + 3x)\overline{w}_3}{x^2\overline{t}^2 + x\overline{t}}$$

*whose denominator* $\overline{D} = \overline{t}(x + x^2\overline{t})$ *is divisible by* $\overline{t}$. *We have* $H = \overline{t}^2(1 + x\overline{t})$ *so* $D_0 = D_1 = 1$ *and* $D_2 = \overline{t}$, *implying that*

$$\omega_1 = 0, \omega_2 = \frac{(1-3x)\overline{t} - 2}{3x\overline{t} + 3}, \text{ and } \omega_3 = \frac{(2-3x)\overline{t} - 1}{3x\overline{t} + 3}$$

*Therefore the equations 2.26 become*

$$0 = b_1' + b_1, 0 = b_2' + \frac{1}{3}b_2, \text{ and } 2x + 3 = b_3' + \frac{2}{3}b_3$$

*whose solutions are* $b_1 = b_2 = 0$ *and* $b_3 = 3x$*, implying that the new integrand is*

$$h = f - \left(\frac{3x\overline{w}_3}{\overline{t}}\right)' = \frac{3}{x}$$

*hence that*

$$\int \frac{3(x+e^x)^{(1/3)} + (2x^2+3x)e^x + 5x^2}{x(x+e^x)^{(1/3)}}\,dx = 3x(x+e^x)^{(2/3)} + 3\int \frac{dx}{x}$$

In the general case when $E$ is not a radical extension of $K(t)$, following the Hermite reduction, any solution of 2.13 must have $v = \sum_{i=1}^{n} v_i w_i / t^m$ where $v_1, \ldots, v_m \in K[t]$. We can compute $v$ by bounding $\deg_t(v_i)$ and comparing the Puiseux expansions at $t = 0$ and at infinity of $\sum_{i=1}^{n} v_i w_i / t^m$ with those of the form 2.20 of the integrand, see [Bron90c, Risc68] for details. Once we are reduced to solving 2.13 for $v \in K$, constants $c_1, \ldots, c_k \in \overline{K}$ and $u_1, \ldots, u_k \in E(c_1, \ldots, c_k)^*$, constants $\rho_1, \ldots, \rho_s \in \overline{K}$ can be determined at all the places above $t = 0$ and at infinity in a manner similar to the algebraic logarithmic case, at which point the algorithm proceeds by constructing the divisors $\delta_j$ and the $u_j$'s as in that case. Again, the details are quite technical and can be found in [Bron90c, Risc68, Risc69a].

# Chapter 3

# Singular Value Decomposition

## 3.1 Singular Value Decomposition Tutorial

When you browse standard web sources like Wikipedia to learn about Singular Value Decomposition [Puff09] or SVD you find many equations, but not an intuitive explanation of what it is or how it works. SVD is a way of factoring matrices into a series of linear approximations that expose the underlying structure of the matrix. Two important properties are that the linear factoring is exact and optimal. Exact means that the series of linear factors, added together, exactly equal the original matrix. Optimal means that, for the standard means of measuring matrix similarity (the Frobenius norm), these factors give the best possible linear approximation at each step in the series.

SVD is extraordinarily useful and has many applications such as data analysis, signal processing, pattern recognition, image compression, weather prediction, and Latent Sematic Analysis or LSA (also referred to as Latent Semantic Indexing). Why is SVD so useful and how does it work?

As a simple example, let's look at golf scores. Suppose Phil, Tiger, and Vijay play together for 9 holes and they each make par on every hole. Their scorecard, which can also be viewed as a (hole x player) matrix might look like this.

| Hole | Par | Phil | Tiger | Vijay |
|---|---|---|---|---|
| 1 | 4 | 4 | 4 | 4 |
| 2 | 5 | 5 | 5 | 5 |
| 3 | 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 | 4 |
| 5 | 4 | 4 | 4 | 4 |
| 6 | 4 | 4 | 4 | 4 |
| 7 | 4 | 4 | 4 | 4 |
| 8 | 3 | 3 | 3 | 3 |
| 9 | 5 | 5 | 5 | 5 |

Let's look at the problem of trying to predict what score each player will make on a given hole. One idea is give each hole a HoleDifficulty factor, and each player a PlayerAbility factor. The actual score is predicted by multiplying these two factors together.

PredictedScore = HoleDifficulty * PlayerAbility

For the first attempt, let's make the HoleDifficulty be the par score for the hole, and let's make the player ability equal to 1. So on the first hole, which is par 4, we would expect a player of ability 1 to get a score of 4.

PredictedScore = HoleDifficulty * PlayerAbility = 4 * 1 = 4

For our entire scorecard or matrix, all we have to do is multiply the PlayerAbility (assumed to be 1 for all players) by the HoleDifficulty (ranges from par 3 to par 5) and we can exactly predict all the scores in our example.

In fact, this is the one dimensional (1-D) SVD factorization of the scorecard. We can represent our scorecard or matrix as the product of two vectors, the HoleDifficulty vector and the PlayerAbility vector. To predict any score, simply multiply the appropriate HoleDifficulty factor by the appropriate PlayerAbility factor. Following normal vector multiplication rules, we can

generate the matrix of scores by multiplying the HoleDifficulty vector by the PlayerAbility vector, according to the following equation.

| Phil | Tiger | Vijay |
|---|---|---|
| 4 | 4 | 4 |
| 5 | 5 | 5 |
| 3 | 3 | 3 |
| 4 | 4 | 4 |
| 4 | 4 | 4 |
| 4 | 4 | 4 |
| 4 | 4 | 4 |
| 3 | 3 | 3 |
| 5 | 5 | 5 |

$=$

| |
|---|
| 4 |
| 5 |
| 3 |
| 4 |
| 4 |
| 4 |
| 4 |
| 3 |
| 5 |

$*$

| Phil | Tiger | Vijay |
|---|---|---|
| 1 | 1 | 1 |

which is HoleDifficulty * PlayerAbility

Mathematicians like to keep everything orderly, so the convention is that all vectors should be scaled so they have length 1. For example, the PlayerAbility vector is modified so that the sum of the squares of its elements add to 1, instead of the current $12 + 12 + 12 = 3$. To do this, we have to divide each element by the square root of 3, so that when we square it, it becomes and the three elements add to 1. Similarly, we have to divide each HoleDifficulty element by the square root of 148. The square root of 3 times the square root of 148 is our scaling factor 21.07. The complete 1-D SVD factorization (to 2 decimal places) is:

| Phil | Tiger | Vijay |
|---|---|---|
| 4 | 4 | 4 |
| 5 | 5 | 5 |
| 3 | 3 | 3 |
| 4 | 4 | 4 |
| 4 | 4 | 4 |
| 4 | 4 | 4 |
| 4 | 4 | 4 |
| 3 | 3 | 3 |
| 5 | 5 | 5 |

$=$

| |
|---|
| 0.33 |
| 0.41 |
| 0.25 |
| 0.33 |
| 0.33 |
| 0.33 |
| 0.33 |
| 0.25 |
| 0.41 |

$*$ 21.07 $*$

| Phil | Tiger | Vijay |
|---|---|---|
| 0.58 | 0.58 | 0.58 |

which is HoleDifficulty * ScaleFactor * PlayerAbility

Our HoleDifficulty vector, that starts with 0.33, is called the Left Singular Vector. The ScaleFactor is the Singular Value, and our PlayerAbility vector, that starts with 0.58 is the Right Singular Vector. If we represent these 3 parts exactly, and multiply them together,

we get the exact original scores. This means our matrix is a rank 1 matrix, another way of saying it has a simple and predictable pattern.

More complicated matrices cannot be completely predicted just by using one set of factors as we have done. In that case, we have to introduce a second set of factors to refine our predictions. To do that, we subtract our predicted scores from the actual scores, getting the residual scores. Then we find a second set of HoleDifficulty2 and PlayerAbility2 numbers that best predict the residual scores.

Rather than guessing HoleDifficulty and PlayerAbility factors and subtracting predicted scores, there exist powerful algorithms than can calculate SVD factorizations for you. Let's look at the actual scores from the first 9 holes of the 2007 Players Championship as played by Phil, Tiger, and Vijay.

| Hole | Par | Phil | Tiger | Vijay |
|---|---|---|---|---|
| 1 | 4 | 4 | 4 | 5 |
| 2 | 5 | 4 | 5 | 5 |
| 3 | 3 | 3 | 3 | 2 |
| 4 | 4 | 4 | 5 | 4 |
| 5 | 4 | 4 | 4 | 4 |
| 6 | 4 | 3 | 5 | 4 |
| 7 | 4 | 4 | 4 | 3 |
| 8 | 3 | 2 | 4 | 4 |
| 9 | 5 | 5 | 5 | 5 |

The 1-D SVD factorization of the scores is shown below. To make this example easier to understand, I have incorporated the ScaleFactor into the PlayerAbility and HoleDifficulty vectors so we can ignore the ScaleFactor for this example.

| Phil | Tiger | Vijay |
|---|---|---|
| 3.95 | 4.64 | 4.34 |
| 4.27 | 5.02 | 4.69 |
| 2.42 | 2.85 | 2.66 |
| 3.97 | 4.67 | 4.36 |
| 3.64 | 4.28 | 4.00 |
| 3.69 | 4.33 | 4.05 |
| 3.33 | 3.92 | 3.66 |
| 3.08 | 3.63 | 3.39 |
| 4.55 | 5.35 | 5.00 |

=

| |
|---|
| 4.34 |
| 4.69 |
| 2.66 |
| 4.36 |
| 4.00 |
| 4.05 |
| 3.66 |
| 3.39 |
| 5.00 |

*

| Phil | Tiger | Vijay |
|---|---|---|
| 0.91 | 1.07 | 1.00 |

which is HoleDifficulty * PlayerAbility

Notice that the HoleDifficulty factor is almost the average of that hole for the 3 players. For example hole 5, where everyone scored 4, does have a factor of 4.00. However hole 6, where the average score is also 4, has a factor of 4.05 instead of 4.00. Similarly, the PlayerAbility is almost the percentage of par that the player achieved, For example Tiger shot 39 with par being 36, and 39/36 = 1.08 which is almost his PlayerAbility factor (for these 9 holes) of 1.07.

Why don't the hole averages and par percentages exactly match the 1-D SVD factors? The answer is that SVD further refines those numbers in a cycle. For example, we can start by assuming HoleDifficulty is the hole average and then ask what PlayerAbility best matches the scores, given those HoleDifficulty numbers? Once we have that answer we can go back and ask what HoleDifficulty best matches the scores given those PlayerAbility numbers? We keep iterating this way until we converge to a set of factors that best predict the score. SVD

shortcuts this process and immediately give us the factors that we would have converged to if we carried out the process.

One very useful property of SVD is that it always finds the optimal set of factors that best predict the scores, according to the standard matrix similarity measure (the Frobenius norm). That is, if we use SVD to find the factors of a matrix, those are the best factors that can be found. This optimality property means that we don't have to wonder if a different set of numbers might predict scores better.

Now let's look at the difference between the actual scores and our 1-D approximation. A plus difference means that the actual score is higher than the predicted score, a minus difference means the actual score is lower than the prediction. For example, on the first hole Tiger got a 4 and the predicted score was 4.64 so we get $4 - 4.64 = -0.64$. In other words, we must add -0.64 to our prediction to get the actual score.

Once these differences have been found, we can do the same thing again and predict these differences using the formula HoleDifficulty2 * PlayerAbility2. Since these factors are trying to predict the differences, they are the 2-D factors and we have put a 2 after their names (ex. HoleDifficulty2) to show they are the second set of factors.

| Phil | Tiger | Vijay |
|---|---|---|
| 0.05 | -0.64 | 0.66 |
| -0.28 | -0.02 | 0.31 |
| 0.58 | 0.15 | -0.66 |
| 0.03 | 0.33 | -0.36 |
| 0.36 | -0.28 | 0.00 |
| -0.69 | 0.67 | -0.05 |
| 0.67 | 0.08 | -0.66 |
| -1.08 | 0.37 | 0.61 |
| 0.45 | -0.35 | 0.00 |

=

| |
|---|
| -0.18 |
| -0.38 |
| 0.80 |
| 0.15 |
| 0.35 |
| -0.67 |
| 0.89 |
| -1.29 |
| 0.44 |

*

| Phil | Tiger | Vijay |
|---|---|---|
| 0.82 | -0.20 | -0.53 |

which is HoleDifficulty(2) * PlayerAbility(2)

There are some interesting observations we can make about these factors. Notice that hole 8 has the most significant HoleDifficulty2 factor (1.29). That means that it is the hardest hole to predict. Indeed, it was the only hole on which none of the 3 players made par. It was especially hard to predict because it was the most difficult hole relative to par ($HoleDifficulty - par$) = $(3.39 - 3) = 0.39$, and yet Phil birdied it making his score more than a stroke below his predicted score (he scored 2 versus his predicted score of 3.08). Other holes that were hard to predict were holes 3 (0.80) and 7 (0.89) because Vijay beat Phil on those holes even though, in general, Phil was playing better.

The full SVD for this example matrix (9 holes by 3 players) has 3 sets of factors. In general, a m x n matrix where m ¿= n can have at most n factors, so our $9x3$ matrix cannot have more than 3 sets of factors. Here is the full SVD factorization (to two decimal places).

| Phil | Tiger | Vijay |
|---|---|---|
| 4 | 4 | 5 |
| 4 | 5 | 5 |
| 3 | 3 | 2 |
| 4 | 5 | 4 |
| 4 | 4 | 4 |
| 3 | 5 | 4 |
| 4 | 4 | 3 |
| 2 | 4 | 4 |
| 5 | 5 | 5 |

=

| | | |
|---|---|---|
| 4.34 | -0.18 | -0.90 |
| 4.69 | -0.38 | -0.15 |
| 2.66 | 0.80 | 0.40 |
| 4.36 | 0.15 | 0.47 |
| 4.00 | 0.35 | -0.29 |
| 4.05 | -0.67 | 0.68 |
| 3.66 | 0.89 | 0.33 |
| 3.39 | -1.29 | 0.14 |
| 5.00 | 0.44 | -0.36 |

*

| Phil | Tiger | Vijay |
|---|---|---|
| 0.91 | 1.07 | 1.00 |
| 0.82 | -0.20 | -0.53 |
| -0.21 | 0.76 | -0.62 |

which is HoleDifficulty(1-3) * PlayerAbility(1-3)

By SVD convention, the HoleDifficulty and PlayerAbility vectors should all have length 1, so the conventional SVD factorization is:

| Phil | Tiger | Vijay |
|---|---|---|
| 4 | 4 | 5 |
| 4 | 5 | 5 |
| 3 | 3 | 2 |
| 4 | 5 | 4 |
| 4 | 4 | 4 |
| 3 | 5 | 4 |
| 4 | 4 | 3 |
| 2 | 4 | 4 |
| 5 | 5 | 5 |

=

| | | |
|---|---|---|
| 0.35 | 0.09 | -0.64 |
| 0.38 | 0.19 | -0.10 |
| 0.22 | -0.40 | 0.28 |
| 0.36 | -0.08 | 0.33 |
| 0.33 | -0.18 | -0.20 |
| 0.33 | 0.33 | 0.48 |
| 0.30 | -0.44 | 0.23 |
| 0.28 | 0.64 | 0.10 |
| 0.41 | -0.22 | -0.25 |

*

| | | |
|---|---|---|
| 21.07 | 0 | 0 |
| 0 | 2.01 | 0 |
| 0 | 0 | 1.42 |

*

| Phil | Tiger | Vijay |
|---|---|---|
| 0.53 | 0.62 | 0.58 |
| -0.82 | 0.20 | 0.53 |
| -0.21 | 0.76 | -0.62 |

which is HoleDifficulty(1-3)* ScaleFactor(1-3) * PlayerAbility(1-3)

We hope that you have some idea of what SVD is and how it can be used. The next section covers applying SVD to Latent Sematic Analysis or LSA. Although the domain is different, the concepts are the same. We are trying to predict patterns of how words occur in documents instead of trying to predict patterns of how players score on holes.

# Chapter 4

# Quaternions

from [Altm05]:

> Quaternions are inextricably linked to rotations. Rotations, however, are an accident of three-dimensional space. In spaces of any other dimensions, the fundamental operations are reflections (mirrors). The quaternion algebra is, in fact, merely a sub-algebra of the Clifford algebra of order three. If the quaternion algebra might be labelled the algebra of rotations, then the Clifford algebra is the algebra of mirrors and it is thus vastly more general than quaternion algebra.

Peter Guthrie Tait, Robert S. Sutor, Timothy Daly

## Preface

The Theory of Quaternions is due to Sir William Rowan Hamilton, Royal Astronomer of Ireland, who presented his first paper on the subject to the Royal Irish Academy in 1843. His Lectures on Quaternions were published in 1853, and his Elements, in 1866, shortly after his death. The Elements of Quaternions by Tait [Tait1890] is the accepted text-book for advanced students.

Large portions of this file are derived from a public domain version of Tait's book combined with the algebra available in Axiom. The purpose is to develop a tutorial introduction to the Axiom domain and its uses.

## 4.1 Quaternions

## 4.2 Vectors, and their Composition

**1**. For at least two centuries the geometrical representation of the negative and imaginary algebraic quantities, $-1$ and $\sqrt{-1}$ has been a favourite subject of speculation with mathematicians. The essence of almost all of the proposed processes consists in employing such expressions to indicate the DIRECTION, not the *length*, of lines.

**2**. Thus it was long ago seen that if positive quantities were measured off in one direction along a fixed line, a useful and lawful convention enabled us to express negative quantities of the same kind by simply laying them off on the same line in the opposite direction. This convention is an essential part of the Cartesian method, and is constantly employed in Analytical Geometry and Applied Mathematics.

**3**. Wallis, towards the end of the seventeenth century, proposed to represent the impossible roots of a quadratic equation by going *out* of the line on which, if real, they would have been laid off. This construction is equivalent to the consideration of $\sqrt{-1}$ as a directed unit-line perpendicular to that on which real quantities are measured.

**4**. In the usual notation of Analytical Geometry of two dimensions, when rectangular axes are employed, this amounts to reckoning each unit of length along $Oy$ as $+\sqrt{-1}$, and on $Oy'$ as $-\sqrt{-1}$ ; while on $Ox$ each unit is $+1$, and on $Ox$ it is $-1$.

If we look at these four lines in circular order, i.e. in the order of positive rotation (that of the northern hemisphere of the earth about its axis, or *opposite* to that of the hands of a watch), they give

$$1, \sqrt{-1}, -1, -\sqrt{-1}$$

In Axiom the same elements would be written as complex numbers which are constructed using the function **complex**:

```
complex(1,0)
```

$$1$$

```
                                                    Type:  Complex Integer
```

```
complex(0,1)
```

$$\%i$$

```
                                                    Type:  Complex Integer
```

```
complex(-1,0)
```

$$-1$$

```
                                                    Type:  Complex Integer
```

```
complex(0,-1)
```

$$-i$$

```
                                                    Type:  Complex Integer
```

Note that %i is of type Complex(Integer), that is, the imaginary part of a complex number. The apparently equivalent expression
`sqrt(-1)`

$$\sqrt{-1}$$

Type: AlgebraicNumber

has the type AlgebraicNumber which means that it is the root of a polynomial with rational coefficients.

In this series each expression is derived from that which precedes it by multiplication by the factor $\sqrt{-1}$. Hence we may consider $\sqrt{-1}$ as an operator, analogous to a handle perpendicular to the plane of $xy$, whose effect on any line in that plane is to make it rotate (positively) about the origin through an angle of 90°.

In Axiom
`%i*%i`

$$-1$$

Type: Complex Integer

**5**. In such a system, (which seems to have been first developed, in 1805, by Buée) a point in the plane of reference is defined by a single imaginary expression. Thus $a + b\sqrt{-1}$ may be considered as a single quantity, denoting the point, $P$, whose coordinates are $a$ and $b$. Or, it may be used as an expression for the line $OP$ joining that point with the origin. In the latter sense, the expression $a + b\sqrt{-1}$ implicitly contains the *direction*, as well as the *length*, of this line ; since, as we see at once, the direction is inclined at an angle $\tan^{-1}(b/a)$ to the axis of $x$, and the length is $\sqrt{a^2 + b^2}$. Thus, say we have

$$OP = a + b\sqrt{-1}$$

the line $OP$ considered as that by which we pass from one extremity, $O$, to the other, $P$. In this sense it is called a VECTOR. Considering, in the plane, any other vector,

$$OQ = a' + b'\sqrt{-1}$$

In order to created superscripted variables we use the superscript function from the SYMBOL domain. So we can create $a'$ as "ap" (that is, "a-prime") and $b'$ as "bp" ("b-prime") thus (also note that the underscore character is Axiom's escape character which removes any special meaning of the next character, in this case, the quote character):
`ap:=superscript(a,[_'])`

$$a'$$

Type: Symbol

`bp:=superscript(b,[_'])`

$$b'$$

Type: Symbol

at this point we can type
`ap+bp*%i`

$$a' + b'\ \%i$$

Type: Complex Polynomial Integer

the addition of these two lines obviously gives

$$OR = a + a' + (b + b')\sqrt{-1}$$

In Axiom the computation looks like:

```
op:=complex(a,b)
```

$$a + b\ \%i$$

```
                                    Type:  Complex Polynomial Integer
```

```
oq:=complex(ap,bp)
```

$$a' + b'\ \%i$$

```
                                    Type:  Complex Polynomial Integer
```

```
op + oq
```

$$a + a' + (b + b')\%i$$

```
                                    Type:  Complex Polynomial Integer
```

and we see that the sum is the diagonal of the parallelogram on $OP$, $OQ$. This is the law of the composition of simultaneous velocities; and it contains, of course, the law of subtraction of one directed line from another.

**6**. Operating on the first of these symbols by the factor $\sqrt{-1}$, it becomes $-b + a\sqrt{-1}$; and now, of course, denotes the point whose $x$ and $y$ coordinates are $-b$ and $a$; or the line joining this point with the origin. The length is still $\sqrt{a^2 + b^2}$, but the angle the line makes with the axis of $x$ is $\tan^{-1}(-a/b)$; which is evidently greater by $\pi/2$ than before the operation.

```
op*complex(0,1)
```

$$-b + a\ i$$

```
                                    Type:  Complex Polynomial Integer
```

**7**. De Moivre's Theorem tends to lead us still further in the same direction. In fact, it is easy to see that if we use, instead of $\sqrt{-1}$, the more general factor $\cos\alpha + \sqrt{-1}\sin\alpha$, its effect on any line is to turn it through the (positive) angle $\alpha$. in the plane of $x$, $y$. [Of course the former factor, $\sqrt{-1}$, is merely the particular case of this, when $\alpha = \frac{\pi}{2}$].

Thus

$$\begin{array}{rcl} & & (\cos\alpha + \sqrt{-1}\sin\alpha)(a + b\sqrt{-1}) \\ = & & a\cos\alpha - b\sin\alpha + \sqrt{-1}(a\sin\alpha + b\cos\alpha) \end{array}$$

by direct multiplication. The reader will at once see that the new form indicates that a rotation through an angle $\alpha$ has taken place, if he compares it with the common formulae for turning the coordinate axes through a given angle. Or, in a less simple manner, thus

$$\begin{array}{rcl} \mathit{Length} & = & \sqrt{(a\cos\alpha - b\sin\alpha)^2 + (a\sin\alpha + b\cos\alpha)^2} \\ & = & \sqrt{a^2 + b^2} \end{array}$$

as before.

Inclination to axis of $x$

$$\begin{array}{rcl} & = & \tan^{-1}\frac{a\sin\alpha + b\cos\alpha}{a\cos\alpha - b\sin\alpha} \\ & = & \tan^{-1}\frac{\tan\alpha + \frac{b}{a}}{1 - \frac{b}{a}\tan\alpha} \\ & = & \alpha + \tan^{-1}\frac{b}{a} \end{array}$$

**8**. We see now, as it were, why it happens that

$$(\cos\alpha + \sqrt{-1}\sin\alpha)^m = \cos m\alpha + \sqrt{-1}\sin m\alpha$$

In fact, the first operator produces $m$ successive rotations in the same direction, each through the angle $\alpha$ ; the second, a single rotation through the angle $m\alpha$.

**9**. It may be interesting, at this stage, to anticipate so far as to remark that in the theory of Quaternions the analogue of

$$\begin{array}{ll} & \cos\theta + \sqrt{-1}\sin\theta \\ \text{is} & \cos\theta + \omega\sin\theta \\ \text{where} & \omega^2 = -1 \end{array}$$

Here, however, $\omega$ is not the algebraic $\sqrt{-1}$, but is *any directed unit-line* whatever in space.

**10**. In the present century Argand, Warren, Mourey, and others, extended the results of Wallis and Buée. They attempted to express as a line the product of two lines each represented by a symbol such $a + b\sqrt{-1}$. To a certain extent they succeeded, but all their results remained confined to two dimensions.

The product, $\prod$, of two such lines was defined as the fourth proportional to unity and the two lines, thus

$$\begin{array}{ll} & 1 : a + b\sqrt{-1} :: a' + b'\sqrt{-1} : \prod \\ \text{or} & \prod = (aa' - bb') + (a'b + b'a)\sqrt{-1} \end{array}$$

The length of $\prod$ is obviously the product of the lengths of the factor lines; and its direction makes an angle with the axis of $x$ which is the sum of those made by the factor lines. From this result the quotient of two such lines follows immediately.

**11**. A very curious speculation, due to Servois and published in 1813 in Gergonne's *Annales*, is one of the very few, so far as has been discovered, in which a well-founded guess at a possible mode of extension to three dimensions is contained. Endeavouring to extend to *space* the form $a + b\sqrt{-1}$ for the plane, he is guided by analogy to write for a directed unit-line in space the form

$$p\cos\alpha + q\cos\beta + r\cos\gamma$$

where $\alpha$, $\beta$, $\gamma$ are its inclinations to the three axes. He perceives easily that $p$, $q$, $r$ must be *non-reals*: but, he asks, "seraient-elles *imaginaires* réductibles à la forme générale $A + B\sqrt{-1}$?" The $i$,$j$, $k$ of the Quaternion Calculus furnish an answer to this question. (See Chap. II.) But it may be remarked that, in applying the idea to lines in a plane, a vector $OP$ will no longer be represented (as in §5) by

$$\begin{array}{lrcl} & \mathrm{OP} & = & a + b\sqrt{-1} \\ \text{but by} & \mathrm{OP} & = & pa + qb \\ \text{And if, similarly,} & \mathrm{OQ} & = & pa' + qb' \end{array}$$

the addition of these two lines gives for $OR$ (which retains its previous signification)

$$OR = p(a + a' + q(b + b')$$

**12**. Beyond this, few attempts were made, or at least recorded, in earlier times, to extend the principle to space of three dimensions; and, though many such had been made before 1843, none, with the single exception of Hamilton's, have resulted in simple, practical methods; all, however ingenious, seeming to lead almost at once to processes and results of fearful complexity.

For a lucid, complete, and most impartial statement of the claims of his predecessors in this field we refer to the Preface to Hamilton's *Lectures on Quaternions*. He there shows how his long protracted investigations of Sets culminated in this unique system of tridimensional-space geometry.

**13**. It was reserved for Hamilton to discover the use and properties of a class of symbols which, though all in a certain sense square roots of -1, may be considered as *real* unit lines, tied down to no particular direction in space ; the expression for a vector is, or may be taken to be,

$$\rho = ix + jy + kz$$

but such vector is considered in connection with an *extraspatial* magnitude $w$, and we have thus the notion of a QUATERNION

$$w + \rho$$

This is the fundamental notion in the singularly elegant, and enormously powerful, Calculus of Quaternions.

While the schemes for using the algebraic $\sqrt{-1}$ to indicate direction make one direction in space expressible by real numbers, the remainder being imaginaries of some kind, and thus lead to expressions which are heterogeneous ; Hamilton s system makes all directions in space equally imaginary, or rather equally real, thereby ensuring to his Calculus the power of dealing with space indifferently in all directions.

In fact, as we will see, the Quaternion method is independent of axes or any supposed directions in space, and takes its reference lines solely from the problem it is applied to.

**14**. But, for the purpose of elementary exposition, it is best to begin by assimilating it as closely as we can to the ordinary Cartesian methods of Geometry of Three Dimensions, with which the student is supposed to be, to some extent at least, acquainted. Such assistance, it will be found, can (as a rule) soon be dispensed with; and Hamilton regarded any apparent necessity for an oc casional recurrence to it, in higher applications, as an indication of imperfect development in the proper methods of the new Calculus.

We commence, therefore, with some very elementary geometrical ideas, relating to the theory of vectors in space. It will subsequently appear how we are thus led to the notion of a Quaternion.

**15**. Suppose we have two points $A$ and $B$ in *space*, and suppose $A$ given, on how many numbers does $B$'s relative position depend ?

If we refer to Cartesian coordinates (rectangular or not) we find that the data required are the excesses of $B$'s three coordinates over those of $A$. Hence three numbers are required.

Or we may take polar coordinates. To define the moon's position with respect to the earth we must have its Geocentric Latitude and Longitude, or its Right Ascension and Declination, and, in addition, its distance or radius-vector. *Three* again.

**16**. Here it is to be carefully noticed that nothing has been said of the *actual* coordinates

of either A or B, or of the earth and moon, in space; it is only the *relative* coordinates that are contemplated.

Hence any expression, as $\overline{AB}$, denoting a line considered with reference to direction and currency as well as length, (whatever may be its actual position in space) contains implicitly *three* numbers, and all lines parallel and equal to $AB$, and concurrent with it, depend in the same way upon the same three. Hence, *all lines which are equal, parallel, and concurrent, may be represented by a common symbol, and that symbol contains three distinct numbers.* In this sense a line is called a VECTOR, since by it we pass from the one extremity, $A$, to the other, $B$, and it may thus be considered as an instrument which *carries* $A$ to $B$: so that a vector may be employed to indicate a definite *translation* in space.

[The term " currency " has been suggested by Cayley for use instead of the somewhat vague suggestion sometimes taken to be involved in the word "direction." Thus parallel lines have the same direction, though they may have similar or opposite currencies. The definition of a vector essentially includes its currency.]

**17**. We may here remark, once for all, that in establishing a new Calculus, we are at liberty to give any definitions whatever of our symbols, provided that no two of these interfere with, or contradict, each other, and in doing so in Quaternions sl simplicity and (so to speak) *naturalness* were the inventor's aim.

**18**. Let $\overline{AB}$ be represented by $\alpha$, we know that $\alpha$ involves *three* separate numbers, and that these depend solely upon the position of $B$ *relatively* to $A$. Now if $CD$ be equal in length to $AB$ and if these lines be parallel, and have the same currency, we may evidently write

$$\overline{CD} = \overline{AB} = \alpha$$

where it will be seen that the sign of equality between vectors contains implicitly *equality in length*, *parallelism in direction*, and *concurrency*. So far we have *extended* the meaning of an algebraical symbol. And it is to be noticed that an equation between vectors, as

$$\alpha = \beta$$

contains *three* distinct equations between mere numbers.

**19**. We must now define + (and the meaning of − will follow) in the new Calculus. Let $A$, $B$, $C$ be any three points, and (with the above meaning of = ) let

$$\overline{AB} = \alpha, \overline{BC} = \beta, \overline{AC} = \gamma$$

If we define + (in accordance with the idea (§16) that a vector represents a *translation*) by the equation

$$\alpha + \beta = \gamma$$

$$\text{or} \qquad \overline{AB} + \overline{BC} = \overline{AC}$$

we contradict nothing that precedes, but we at once introduce the idea that *vectors are to be compounded, in direction and magnitude, like simultaneous velocities.* A reason for this may be seen in another way if we remember that by *adding* the (algebraic) differences of the Cartesian coordinates of $B$ and $A$, to those of the coordinates of $C$ and $B$, we get those of the coordinates of $C$ and $A$. Hence these coordinates enter *linearly* into the expression for a vector. (See, again, §5.)

**20**. But we also see that if $C$ and $A$ coincide (and $C$ may be *any* point)

$$\overline{AC} = 0$$

for no vector is then required to carry $A$ to $C$. Hence the above relation may be written, in this case,

$$\overline{AB} + \overline{BA} = 0$$

or, introducing, and by the same act defining, the symbol $-$,

$$\overline{AB} = -\overline{BA}$$

Hence, *the symbol $-$, applied to a vector, simply shows that its currency is to be reversed.* And this is consistent with all that precedes; for instance,

$$\begin{array}{lrcl} & \overline{AB} + \overline{BC} & = & \overline{AC} \\ \text{and} & \overline{AB} = \overline{AC} & - & \overline{BC} \\ \text{or} & = \overline{AC} & + & \overline{CB} \end{array}$$

are evidently but different expressions of the same truth.

**21**. In any triangle, $ABC$, we have, of course,

$$\overline{AB} + \overline{BC} + \overline{CA} = 0$$

and, in any closed polygon, whether plane or gauche,

$$\overline{AB} + \overline{BC} + \ldots + \overline{YZ} + \overline{ZA} = 0$$

In the case of the polygon we have also

$$\overline{AB} + \overline{BC} + \ldots + \overline{YZ} = \overline{AZ}$$

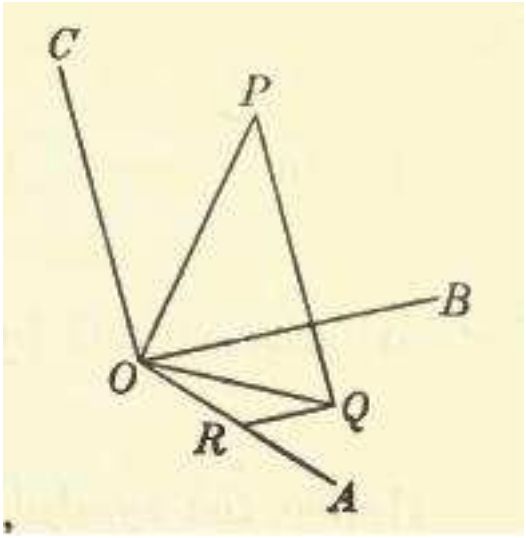
These are the well-known propositions regarding composition of velocities, which, by Newton's second law of motion, give us the geometrical laws of composition of forces acting at one point.

**22**. If we compound any number of **parallel** vectors, the result is obviously a numerical multiple of any one of them. Thus, if $A$, $B$, $C$ are in one straight line,

$$\overline{BC} = x\overline{AB}$$

where $x$ is a number, positive when $B$ lies between $A$ and $C$, otherwise negative; but such that its numerical value, independent of sign, is the ratio of the length of $BC$ to that of $AB$. This is at once evident if $AB$ and $BC$ be commensurable; and is easily extended to incommensurables by the usual *reductio ad absurdum*.

**23**. An important, but almost obvious, proposition is that *any vector may be resolved, and in one way only, into three components parallel respectively to any three given vectors, no two of which are parallel, and which are not parallel to one plane.*




Let $OA$, $OB$, $OC$ be the three fixed vectors, $OP$ any other vector. From $P$ draw $PQ$ parallel to $CO$, meeting the plane $BOA$ in $Q$. [There must be a definite point $Q$, else $PQ$, and therefore $CO$, would be parallel to $BOA$, a case specially excepted.] From $Q$ draw $QR$ parallel to $BO$, meeting $OA$ in $R$.

Then we have $\overline{OP} = \overline{OR} + \overline{RQ} + \overline{QP}$ (§21), and these components are respectively parallel to the three given vectors. By §22 we may express $\overline{OR}$ as a numerical multiple of $\overline{OA}$, $\overline{RQ}$ of $\overline{OB}$, and $\overline{QP}$ of $\overline{OC}$. Hence we have, generally, for any vector in terms of three fixed non-coplanar vectors, $\alpha$, $\beta$, $\gamma$

$$\overline{OP} = \rho = x\alpha + y\beta + z\gamma$$

which exhibits, in one form, the *three* numbers on which a vector depends (§16). Here $x$, $y$, $z$ are perfectly definite, and can have but single values.

**24**. Similarly any vector, as $\overline{OQ}$, in the same plane with $\overline{OA}$ and $\overline{OB}$, can be resolved (in one way only) into components $\overline{OR}$, $\overline{RQ}$, parallel respectively to $\overline{OA}$ and $\overline{OB}$; so long, at least, as these two vectors are not parallel to each other.

**25**. There is particular advantage, in certain cases, in employing a series of *three mutually perpendicular unit-vectors* as lines of reference. This system Hamilton denotes by $i$,$j$, $k$.

Any other vector is then expressible as

$$\rho = xi + yj + zk$$

Since $i$, $j$, $k$ are unit-vectors, $x$, $y$, $z$ are here the lengths of conterminous edges of a rectangular parallelepiped of which $\rho$ is the vector-diagonal; so that the length of $\rho$ is, in this case,

$$\sqrt{x^2 + y^2 + z^2}$$

Let $$\omega = \xi i + \eta j + \zeta k$$
be any other vector, then (by the proposition of §23) the vector

equation $\rho = \omega$

obviously involves the following three equations among numbers,

$$x = \xi, y = \eta, z = \zeta$$

Suppose $i$ to be drawn eastwards, $j$ northwards, and $k$ upwards, this is equivalent merely to saying that *if two points coincide, they are equally to the east (or west) of any third point, equally to the north (or south) of it, and equally elevated above (or depressed below) its level.*

**26**. It is to be carefully noticed that it is only when $\alpha$, $\beta$, $\gamma$ are not coplanar that a vector equation such as

$$\rho = \omega$$

or $$x\alpha + y\beta + z\gamma = \xi\alpha + \eta\beta + \zeta\gamma$$

necessitates the three numerical equations

$$x = \xi, y = \eta, z = \zeta$$

For, if $\alpha$, $\beta$, $\gamma$ be coplanar (§24), a condition of the following form must hold

$$\gamma = a\alpha + b\beta$$

Hence, $$\rho = (x + za)\alpha + (y + zb)\beta$$
$$\omega = (\xi + \zeta a)\alpha + (\eta + \zeta b)\beta$$

and the equation $$\rho = \omega$$

now requires only the two numerical conditions

$$x + za = \xi + \zeta a \qquad y + zb = \eta + \zeta b$$

**27**. *The Commutative and Associative Laws hold in the combination of vectors by the signs* $+$ *and* $-$. It is obvious that, if we prove this for the sign $+$, it will be equally proved for $-$, because $-$ before a vector (§20) merely indicates that it is to be reversed before being considered positive.

Let $A$, $B$, $C$, $D$ be, in order, the corners of a parallelogram ; we have, obviously,

$$\overline{AB} = \overline{DC} \qquad \overline{AD} = \overline{BC}$$

And $$\overline{AB} + \overline{BC} = \overline{AC} = \overline{AD} + \overline{DC} = \overline{BC} + \overline{AB}$$

Hence the commutative law is true for the addition of any two vectors, and is therefore generally true.

Again, whatever four points are represented by $A$, $B$, $C$, $D$, we

$$\overline{AD} = \overline{AB} + \overline{BD} = \overline{AC} + \overline{CD}$$

or substituting their values for $\overline{AD}$, $\overline{BD}$, $\overline{AC}$ respectively, in these three expressions,

$$\overline{AB} + \overline{BC} + \overline{CD} = \overline{AB} + (\overline{BC} + \overline{CD}) = (\overline{AB} + \overline{BC}) + \overline{CD}$$

And thus the truth of the associative law is evident.

**28**. The equation

$$\rho = x\beta$$

where $\rho$ is the vector connecting a variable point with the origin, $\beta$ a definite vector, and $x$ an indefinite number, represents the straight line drawn from the origin parallel to $\beta$ (§22).

The straight line drawn from $A$, where $\overline{OA} = \alpha$, and parallel to $\beta$, has the equation

$$\rho = \alpha + x\beta \tag{4.1}$$

In words, we may pass directly from $O$ to $P$ by the vector $\overline{OP}$ or $\rho$; or we may pass first to $A$, by means of $\overline{OA}$ or $\alpha$, and then to $P$ along a vector parallel to $\beta$ (§16).

Equation 4.1 is one of the many useful forms into which Quaternions enable us to throw the general equation of a straight line in space. As we have seen (§25) it is equivalent to three numerical equations; but, as these involve the indefinite quantity $x$, they are virtually equivalent to but *two*, as in ordinary Geometry of Three Dimensions.

**29**. A good illustration of this remark is furnished by the fact that the equation

$$\rho = y\alpha + x\beta$$

which contains two indefinite quantities, is virtually equivalent to only one numerical equation. And it is easy to see that it represents the plane in which the lines $\alpha$ and $\beta$ lie; or the surface which is formed by drawing, through every point of $OA$, a line parallel to $OB$. In fact, the equation, as written, is simply §24 in symbols.

And it is evident that the equation

$$\rho = \gamma + y\alpha + x\beta$$

is the equation of the plane passing through the extremity of $\gamma$, and parallel to $\alpha$ and $\beta$.

It will now be obvious to the reader that the equation

$$\rho = p_1\alpha_1 + p_2\alpha_2 + \ldots = \sum p\alpha$$

where $\alpha_1$, $\alpha_2$ , &c. are given vectors, and $p_1$, $p_2$, &c. numerical quantities, *represents a straight line* if $p_1$, $p_2$, &c. be linear functions of *one* indeterminate number; and a *plane*, if they be linear expressions containing two indeterminate numbers. Later (§31 (l)), this theorem will be much extended.

Again, the equation

$$\rho = x\alpha + y\beta + z\gamma$$

refers to *any* point whatever in space, provided $\alpha$, $\beta$, $\gamma$ are not coplanar. (Ante, §23)

**30**. The equation of the line joining any two points $A$ and $B$, where $\overline{OA} = \alpha$ and $\overline{OB} = \beta$, is obviously

$$\rho = \alpha + x(\beta - \alpha)$$

or
$$\rho = \beta + y(\alpha - \beta)$$

These equations are of course identical, as may be seen by putting $1 - y$ for $x$.

The first may be written

$$\rho + (x - 1)\alpha - x\beta = 0$$

or
$$p\rho + q\alpha + r\beta = 0$$

subject to the condition $p + q + r = 0$ identically. That is – A homogeneous linear function of three vectors, equated to zero, expresses that the extremities of these vectors are in one straight line, *if the sum of the coefficients be identically zero.*

Similarly, the equation of the plane containing the extremities $A$, $B$, $C$ of the three non-coplanar vectors $\alpha$, $\beta$, $\gamma$ is

$$\rho = \alpha + x(\beta - \alpha) + y(\gamma - \beta)$$

where $x$ and $y$ are each indeterminate.

This may be written

$$p\rho + q\alpha + r\beta + s\gamma = 0$$

with the identical relation

$$p + q + r + x = 0$$

which is one form of the condition that four points may lie in one plane.

**31**. We have already the means of proving, in a very simple manner, numerous classes of propositions in plane and solid geometry. A very few examples, however, must suffice at this stage; since we have hardly, as yet, crossed the threshold of the subject, and are dealing with mere linear equations connecting two or more vectors, and even with them *we are restricted as yet to operations of mere addition*. We will give these examples with a painful minuteness of detail, which the reader will soon find to be necessary only for a short time, if at all.

(a) *The diagonals of a parallelogram bisect each other.*

Let $ABCD$ be the parallelogram, $O$ the point of intersection of its diagonals. Then

$$\overline{AO} + \overline{OB} = \overline{AB} = \overline{DC} = \overline{DO} + \overline{OC}$$

which gives $\overline{AO} - \overline{OC} = \overline{DO} - \overline{OB}$

The two vectors here equated are parallel to the diagonals respectively. Such an equation is, of course, absurd unless

1. The diagonals are parallel, in which case the figure is not a parallelogram;
2. $\overline{AO} = \overline{OC}$, and $\overline{DO} = \overline{OB}$, the proposition.

(b) *To shew that a triangle can be constructed, whose sides are parallel, and equal, to the bisectors of the sides of any triangle.*

Let $ABC$ be any triangle, $Aa$, $Bb$, $Cc$ the bisectors of the sides.

Then

$$\begin{array}{lll} \overline{Aa} & = \overline{AB} + \overline{Ba} & = \overline{AB} + \frac{1}{2}\overline{BC} \\ \overline{Bb} & \ldots & = \overline{BC} + \frac{1}{2}\overline{CA} \\ \overline{Cc} & \ldots & = \overline{CA} + \frac{1}{2}\overline{AB} \end{array}$$

Hence $\overline{Aa} + \overline{Bb} + \overline{Cc} = \frac{3}{2}(\overline{AB} + \overline{BC} + \overline{CA}) = 0$

which (§21) proves the proposition.

Also

$$\begin{array}{rcl} \overline{Aa} & = & \overline{AB} + \frac{1}{2}\overline{BC} \\ & = & \overline{AB} - \frac{1}{2}(\overline{CA} + \overline{AB}) \\ & = & \frac{1}{2}(\overline{AB} - \overline{CA}) \\ & = & \frac{1}{2}(\overline{AB} + \overline{AC}) \end{array}$$

results which are sometimes useful. They may be easily verified by producing $\overline{Aa}$ to twice its length and joining the extremity with $B$.

($b'$) *The bisectors of the sides of a triangle meet in a point, which trisects each of them.*

Taking $A$ as origin, and putting $\alpha$, $\beta$, $\gamma$ for vectors parallel, and equal, to the sides taken in order $BC$, $CA$, $AB$; the equation of $Bb$ is (§28 (1))

$$\rho = \gamma + x(\gamma + \frac{\beta}{2}) = (1+x)\gamma + \frac{x}{2}\beta$$

That of $Cc$ is, in the same way,

$$\rho = -(1+y)\beta - \frac{y}{2}\gamma$$

At the point $O$, where $Bb$ and $Cc$ intersect,

$$\rho = (1+x)\gamma + \frac{x}{2}\beta = -(1+y)\beta - \frac{y}{2}\gamma$$

Since $\gamma$ and $\beta$ are not parallel, this equation gives

$$1 + x = -\frac{y}{2} \quad \text{and} \quad \frac{x}{2} = -(1+y)$$

From these $x = y = -\frac{2}{3}$

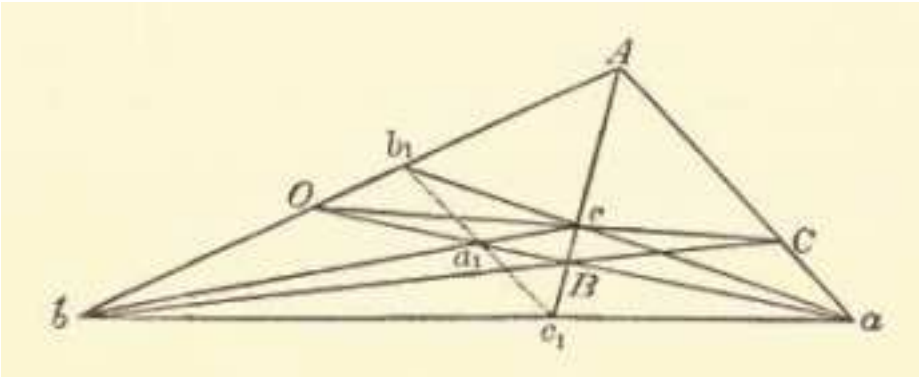
Hence $\overline{AO} = \frac{1}{3}(\gamma - \beta) = \frac{2}{3}\overline{Aa}$ (See Ex. (b))

This equation shows, being a vector one, that $\overline{Aa}$ passes through $O$, and that $AO : Oa :: 2{:}1$.

(c) If

$$\overline{OA} = \alpha$$
$$\overline{OB} = \beta$$
$$\overline{OC} = l\alpha + m\beta$$

*be three given co-planar vectors, c the intersection of AB, OC, and if the lines indicated in the figure be drawn, the points $a_1$,$b_1$,$c_1$ lie in a straight line.*




We see at once, by the process indicated in §30, that

$$\overline{Oc} = \frac{l\alpha + m\beta}{l+m}, \qquad \overline{Ob} = \frac{l\alpha}{1-m}, \qquad \overline{Oa} = \frac{m\beta}{1-l}$$

Hence we easily find

$$\overline{Oa_1} = -\frac{m\beta}{1-l-2m}, \qquad \overline{Ob_1} = -\frac{l\alpha}{1-2l-m}, \qquad \overline{Oc_1} = \frac{-l\alpha + m\beta}{m-l}$$

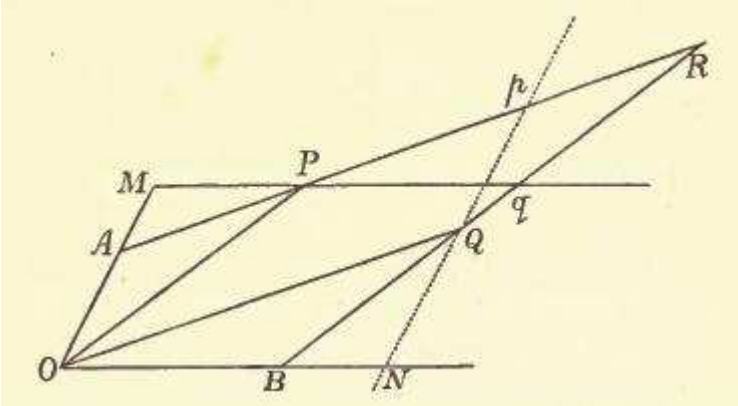
These give

$$-(1-l-2m)\overline{Oa_1} + (1-2l-m)\overline{Ob_1} - (m-l)\overline{Oc_1} = 0$$

But $\quad -(1-l-2m) + (1-2l-m) - (m-l) = 0$ identically.

This, by §30, proves the proposition.

(d) Let $\overline{OA} = \alpha$, $\overline{OB} = \beta$, be any two vectors. If $MP$ be a given line parallel to $OB$; and $OQ$, $BQ$, be drawn parallel to $AP$, $OP$ respectively ; the locus of $Q$ is a straight line parallel to $OA$.




Let $\quad \overline{OM} = e\alpha$

Then $\quad \overline{AP} = \overline{e-1}\alpha + x\beta$

Hence the equation of $OQ$ is

$$\rho = y(\overline{e-1}\alpha + x\beta)$$

and that of $BQ$ is $\quad \rho = \beta + z(e\alpha + x\beta)$

At Q we have, therefore,

$$\left.\begin{aligned} xy &= 1 + zx \\ y(e-1) &= ze \end{aligned}\right\}$$

These give $xy = e$, and the equation of the locus of $Q$ is

$$\rho = e\beta + y'\alpha$$

i.e. a straight line parallel to $OA$, drawn through $N$ in $OB$ produced, so that

$$ON : OB :: OM : OA$$

COR. If $BQ$ meet $MP$ in $q$, $\overline{Pq} = \beta$; and if $AP$ meet $NQ$ in $p$, $\overline{Qp} = \alpha$.

Also, for the point $R$ we have $\overline{pR} = \overline{AP}$, $\overline{QR} = \overline{Bq}$.

Further, the locus of $R$ is a hyperbola, of which $MP$ and $NQ$ are the asymptotes. See, in this connection, §31 (k) below.

Hence, *if from any two points, A and B, lines be drawn intercepting a given length Pq on a given line Mq ; and if, from R their point of intersection, Rp be laid off = PA, and RQ = qB ; Q and p lie on a fixed straight line, and the length of Qp is constant.*

(e) *To find the centre of inertia of any system of masses.*

If $\overline{OA} = \alpha$, $\overline{OB} = \alpha_1$, be the vector sides of any triangle, the vector from the vertex dividing the base $AB$ in $C$ so that

$$BC : CA :: m : m_1$$

is $\frac{m\alpha + m_1\alpha_1}{m+m_1}$

For $AB$ is $\alpha_1 - \alpha$, and therefore $\overline{AC}$ is

$$\frac{m_1}{m+m_1}(\alpha_1 - \alpha)$$

Hence
$$\overline{OC} = \overline{OA} + \overline{AC}$$
$$= \alpha + \frac{m_1}{m+m_1}(\alpha_1 - \alpha)$$
$$= \frac{m\alpha + m_1\alpha_1}{m+m_1}$$

This expression shows how to find the centre of inertia of two masses ; $m$ at the extremity of $\alpha$, $m_1$ at that of $\alpha_1$. Introduce $m_2$ at the extremity of $a_2$, then the vector of the centre of inertia of the three is, by a second application of the formula,

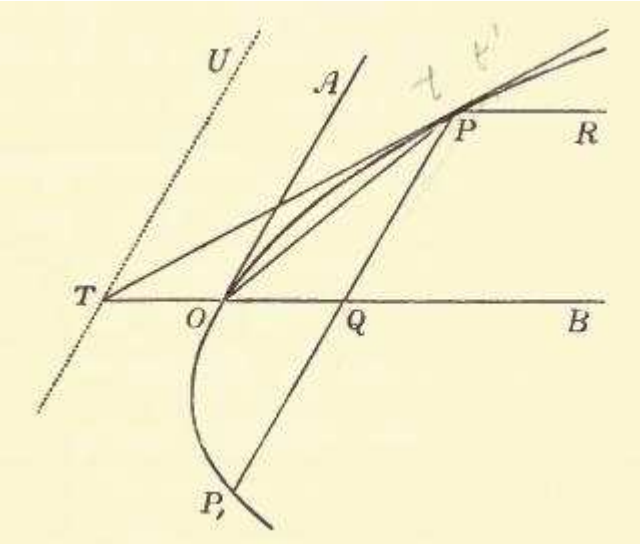
$$\frac{(m+m_1)(\frac{m\alpha+m_1\alpha_1}{m+m_1}) + m_2\alpha_2}{(m+m_1)+m_2} = \frac{m\alpha + m_1\alpha_1 + m_2\alpha_2}{m+m_1+m_2}$$

From this it is clear that, for any number of masses, expressed generally by $m$ at the extremity of the vector $\alpha$, the vector of the centre of inertia is

$$\beta = \frac{\sum(m\alpha)}{\sum(m)}$$

This may be written $\sum m(\alpha - \beta) = 0$
Now a $\alpha_1 - \beta$ is the vector of $m_1$ with respect to the centre of inertia. Hence the theorem, *If the vector of each element of a mass, drawn from the centre of inertia, be increased in length in proportion to the mass of the element, the sum of all these vectors is zero.*

(f) We see at once that the equation




$$\rho = \alpha t + \frac{\beta t^2}{2}$$

where $t$ is an indeterminate number, and $\alpha$, $\beta$ given vectors, represents a parabola. The origin, $O$, is a point on the curve, $\beta$ is parallel to the axis, i.e. is the diameter $OB$ drawn from the origin, and $\alpha$ is $OA$ the tangent at the origin. In the figure

$$\overline{QP} = \alpha t, \qquad \overline{OQ} = \frac{\beta t^2}{2}$$

The secant joining the points where $t$ has the values $t$ and $t'$ is represented by the equation

$$\begin{aligned} \rho &= \alpha t + \tfrac{\beta t^2}{2} + x\left(\alpha t' + \tfrac{\beta t'^2}{2} - \alpha t - \tfrac{\beta t^2}{2}\right) \qquad (\S 30) \\ &= \alpha t + \tfrac{\beta t^2}{2} + x(t'-t)\left\{\alpha + \beta\tfrac{t'-t}{2}\right\} \end{aligned}$$

Write $x$ for $x(t'-t)$ [which may have any value], then put $t' = t$, and the equation of the tangent at the point $(t)$ is

$$\rho = \alpha t + \frac{\beta t^2}{2} + x(\alpha + \beta t)$$

In this put $x = -t$, and we have

$$\rho = -\frac{\beta t^2}{2}$$

or the intercept of the tangent on the diameter is equal in length to the abscissa of the point of contact, but has the opposite currency.

Otherwise: the tangent is parallel to the vector $\alpha + \beta t$ or $\alpha t + \beta t^2$ or $\frac{\beta t^2}{2} + \alpha t + \frac{\beta t^2}{2}$ or $\overline{OQ} + \overline{OP}$. But $\overline{TP} = \overline{TO} + \overline{OP}$, hence $\overline{TO} = \overline{OQ}$.

(g) Since the equation of any tangent to the parabola is

$$\rho = \alpha t + \frac{\beta t^2}{2} + x(\alpha + \beta t)$$

let us find the tangents which can be drawn from a given point. Let the vector of the point be

$$\rho = p\alpha + q\beta \qquad (\S 24)$$

Since the tangent is to pass through this point, we have, as con ditions to determine $t$ and $x$,

$$t + x = p$$
$$\frac{t^2}{2} + xt = q$$

by equating respectively the coefficients of $\alpha$ and $\beta$.

Hence $$t = p \pm \sqrt{p^2 - 2q}$$

Thus, in general, *two* tangents can be drawn from a given point. These coincide if

$$p^2 = 2q$$

that is, if the vector of the point from which they are to be drawn is

$$\rho = p\alpha + q\beta = p\alpha + \frac{p^2}{2}\beta$$

i.e. if the point lies on the parabola. They are imaginary if $2q > p^2$, that is, if the point be

$$\rho = p\alpha + \left(\frac{p^2}{2} + r\right)\beta$$

$r$ being *positive*. Such a point is evidently *within* the curve, as at $R$, where $\overline{OQ} = \frac{p^2}{2}\beta$, $\overline{QP} = p\alpha$, $\overline{PR} = r\beta$.

(h) Calling the values of $t$ for the two tangents found in (g) $t_1$ and $t_2$ respectively, it is obvious that the vector joining the points of contact is

$$\alpha t_1 + \frac{\beta t_1^2}{2} - \alpha t_2 - \frac{\beta t_2^2}{2}$$

which is parallel to $\alpha + \beta\frac{t_1+t_2}{2}$ or, by the values of $t_1$ and $t_2$ in (g),

$$\alpha + p\beta$$

Its direction, therefore, does not depend on $q$. In words, *If pairs of tangents be drawn to a parabola from points of a diameter produced, the chords of contact are parallel to the tangent at the vertex of the diameter.* This is also proved by a former result, for we must have $\overline{OT}$ for each tangent equal to $\overline{QO}$.

(i) The equation of the chord of contact, for the point whose vector is

$$\rho = p\alpha + q\beta$$

is thus $$\rho = \alpha t_1 + \frac{\beta t_1^2}{2} + y(\alpha + p\beta)$$

Suppose this to pass always through the point whose vector is

$$\rho = a\alpha + b\beta$$

Then we must have

$$\left.\begin{array}{rcl} t_1 + y & = & a \\ \frac{t_1^2}{2} + py & = & b \end{array}\right\}$$

or $$t_1 = p \pm \sqrt{p^2 - 2p\alpha + 2\beta}$$

Comparing this with the expression in (g), we have

$$q = pa - b$$

that is, the point from which the tangents are drawn has the vector a straight line (§28 (1)).

The mere form of this expression contains the proof of the usual properties of the pole and polar in the parabola ; but, for the sake of the beginner, we adopt a simpler, though equally general, process.

Suppose $\alpha = 0$. This merely restricts the pole to the particular diameter to which we have referred the parabola. Then the pole is $Q$, where

$$\rho = b\beta$$

and the polar is the line $TU$, for which

$$\rho = -b\beta + p\alpha$$

*Hence the polar of any point is parallel to the tangent at the extremity of the diameter on which the point lies, and its intersection with that diameter is as far beyond the vertex as the pole is within, and vice versa.*

(j) As another example let us prove the following theorem. *If a triangle be inscribed in a parabola, the three points in which the sides are met by tangents at the angles lie in a straight line.*

Since $O$ is any point of the curve, we may take it as one corner of the triangle. Let $t$ and $t_1$ determine the others. Then, if $\omega_1$,$\omega_2$,$\omega_3$ represent the vectors of the points of intersection of the tangents with the sides, we easily find

$$\begin{array}{rcl} \omega_1 & = & \frac{t_1^2}{2t_1 - t}\left(\alpha + \frac{t}{2}\beta\right) \\ \omega_2 & = & \frac{t^2}{2t - t_1}\left(\alpha + \frac{t_1}{2}\beta\right) \\ \omega_3 & = & \frac{tt_1}{t_1 + t}\alpha \end{array}$$

These values give

$$\frac{2t_1 - t}{t_1}\omega_1 - \frac{2t - t_1}{t}\omega_2 - \frac{t_1^2 - t^2}{tt_1}\omega_3 = 0$$

Also

$$\frac{2t_1 - t}{t_1} - \frac{2t - t_1}{t} - \frac{t_1^2 - t^2}{tt_1} = 0$$

identically.

Hence, by §30, the proposition is proved.

(k) Other interesting examples of this method of treating curves will, of course, suggest themselves to the student. Thus

$$\rho = \alpha\cos t + \beta\sin t$$

or

$$\rho = \alpha x + \beta\sqrt{1 - x^2}$$

represents an ellipse, of which the given vectors $\alpha$ and $\beta$ are semiconjugate diameters. If $t$ represent time, the radius-vector of this ellipse traces out equal areas in equal times. [We may anticipate so far as to write the following :

$$2\text{Area} = T\int V\rho d\rho = TV\alpha\beta.\int dt$$

which will be easily understood later.]

Again,

$$\rho = \alpha t + \frac{\beta}{t} \text{ or } \rho = \alpha\tan x + \beta\cot x$$

evidently represents a hyperbola referred to its asymptotes. [If $t$ represent time, the sectorial area traced out is proportional to $\log t$, taken between proper limits.] Thus, also, the equation

$$\rho = \alpha(t + \sin t) + \beta\cos t$$

in which $\alpha$ and $\beta$ are of equal lengths, and at right angles to one another, represents a cycloid. The origin is at the middle point of the axis ($2\beta$) of the curve. [It may be added that, if t represent *time*, this equation shows the motion of the tracing point, provided the generating circle rolls uniformly, revolving at the rate of a radian per second.]

When the lengths of $\alpha$, $\beta$ are not equal, this equation gives the cycloid distorted by elongation of its ordinates or abscissae : *not* a trochoid. The equation of a trochoid may be written

$$\rho = \alpha(et + \sin t) + \beta\cos t$$

$e$ being greater or less than 1 as the curve is prolate or curtate. The lengths of $\alpha$ and $\beta$ are still taken as equal.

But, so far as we have yet gone with the explanation of the calculus, as we are not prepared to determine the lengths or inclinations of vectors, we can investigate only a very limited class of the properties of curves, represented by such equations as those above written.

(l) We may now, in extension of the statement in §29, make the obvious remark that

$$\rho = \sum p\alpha$$

(where, as in §23, the number of vectors, $\alpha$, can always be reduced to *three*, at most) is the equation of a curve in space, if the numbers $p_1$, $p_2$, &c. are functions of one indeterminate. In such a case the equation is sometimes written

$$\rho = \phi(t)$$

But, if $p_1$, $p_2$, &c. be functions of *two* indeterminates, the locus of the extremity of $\rho$ is a *surface*; whose equation is sometimes written

$$\rho = \phi(t, u)$$

[It may not be superfluous to call the reader's attention to the fact that, in these equations, $\phi(t)$ or $\phi(t, u)$ is necessarily a vector expression, since it is equated to a vector, $\rho$.]

(m) Thus the equation

$$\rho = \alpha \cos t + \beta \sin t + \gamma t \tag{4.2}$$

belongs to a helix, while

$$\rho = \alpha \cos t + \beta \sin t + \gamma u \tag{4.3}$$

represents a cylinder whose generating lines are parallel to $\gamma$, and whose base is the ellipse

$$\rho = \alpha \cos t + \beta \sin t$$

The helix above lies wholly on this cylinder.

Contrast with (2) the equation

$$\rho = u(\alpha \cos t + \beta \sin t + \gamma) \tag{3}$$

which represents a cone of the second degree made up, in fact, of all lines drawn from the origin to the ellipse

$$\rho = \alpha \cos t + \beta \sin t + \gamma$$

If, however, we write

$$\rho = u(\alpha \cos t + \beta \sin t + \gamma t)$$

we form the equation of the transcendental cone whose vertex is at the origin, and on which lies the helix (1).

In general

$$\rho = u\phi(t)$$

is the cone whose vertex is the origin, and on which lies the curve

$$\rho = \phi(t)$$

while $\rho = \phi(t) + u\alpha$
is a cylinder, with generating lines parallel to $\alpha$, standing on the same curve as base.

Again, $\rho = p\alpha + q\beta + r\gamma$
with a condition of the form

$$ap^2 + bq^2 + cr^2 = 1$$

belongs to a central surface of the second order, of which $\alpha$, $\beta$, $\gamma$ are the directions of conjugate diameters. If $a$, $b$, $c$ be all positive, the surface is an ellipsoid.

**32**. In Example $(f)$ above we performed an operation equivalent to the differentiation of a vector with reference to a single *numerical* variable of which it was given as an explicit function. As this process is of very great use, especially in quaternion investigations connected with the motion of a particle or point; and as it will afford us an opportunity of making a preliminary step towards overcoming the novel difficulties which arise in quaternion differentiation; we will devote a few sections to a more careful, though very elementary, exposition of it.

**33**. It is a striking circumstance, when we consider the way in which Newton's original methods in the Differential Calculus have been decried, to find that Hamilton was *obliged* to employ them, and not the more modern forms, in order to overcome the characteristic difficulties of quaternion differentiation. Such a thing as *a differential coefficient has absolutely no meaning in quaternions*, except in those special cases in which we are dealing with degraded quaternions, such as numbers, Cartesian coordinates, &c. But a quaternion expression has always a *differential*, which is, simply, what Newton called a *fluxion*.

As with the Laws of Motion, the basis of Dynamics, so with the foundations of the Differential Calculus ; we are gradually coming to the conclusion that Newton s system is the best after all.

**34**. Suppose $\rho$ to be the vector of a curve in space. Then, generally, $\rho$ may be expressed as the sum of a number of terms, each of which is a multiple of a constant vector by a function of some *one* indeterminate; or, as in §31 $(l)$, if $P$ be a point on the curve,

$$\overline{OP} = \rho = \phi(t)$$

And, similarly, if $Q$ be *any other* point on the curve,

$$\overline{OQ} = \rho_1 = \rho + \delta\rho = \phi(t_1) = \phi(t + \delta t)$$

where $\delta t$ is *any number whatever.*

The vector-chord $\overline{PQ}$ is therefore, rigorously,

$$\delta p = \rho_1 - \rho = \phi(t + \delta t) - \phi t$$

**35**. It is obvious that, in the present case, *because the vectors involved in $\phi$ are constant, and their numerical multipliers alone vary*, the expression $\phi(t + \delta t)$ is, by Taylor's Theorem, equivalent to

$$\phi(t) + \frac{d\phi(t)}{dt}\delta t + \frac{d^2\phi(t)}{dt^2}\frac{(\delta t)^2}{1\ .\ 2} + \dots$$

Hence,

$$\delta\rho = \frac{d\phi(t)}{dt}\delta t + \frac{d^2\phi(t)}{dt^2}\frac{(\delta t)^2}{1\ .\ 2} + \&\text{c.}$$

And we are thus entitled to write, when $\delta t$ has been made indefinitely small,

$$\text{Limit}\left(\begin{array}{c}\delta p\\ \delta t\end{array}\right)_{\delta t=0} = \frac{d\rho}{dt} = \frac{d\phi(t)}{dt} = \phi'(t)$$

In such a case as this, then, we are permitted to differentiate, or to form the differential coefficient of, a vector, according to the ordinary rules of the Differential Calculus. But great additional insight into the process is gained by applying Newton's method.

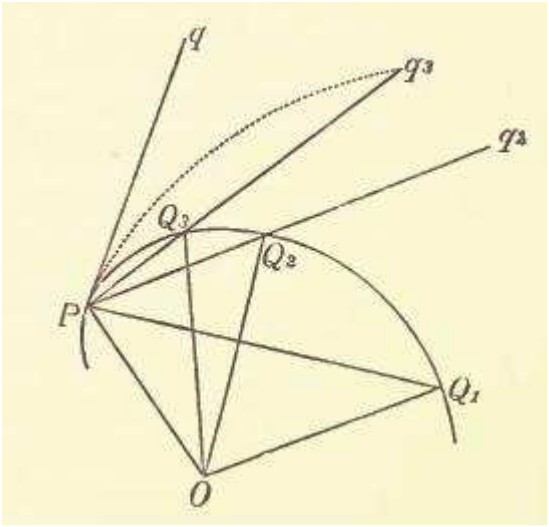
**36**. Let $\overline{OP}$ be

$$\rho = \phi(t)$$

and $overline OQ_1$

$$\rho_1 = \phi(t + dt)$$

where $dt$ is any number whatever.




The number $t$ may here be taken as representing *time*, i.e. we may suppose a point to move along the curve in such a way that the value of $t$ for the vector of the point $P$ of the curve denotes the interval which has elapsed (since a fixed epoch) when the moving point has reached the extremity of that vector. If, then, $dt$ represent any interval, finite or not, we see that

$$\overline{OQ}_1 = \phi(t + dt)$$

will be the vector of the point after the additional interval $dt$.

But this, in general, gives us little or no information as to the velocity of the point at $P$. We will get a better approximation by halving the interval $dt$, and finding $Q_2$ , where $\overline{OQ}_2 = \phi(t + \frac{1}{2}dt)$, as the position of the moving point at that time. Here the vector virtually described in $\frac{1}{2}dt$ is $\overline{PQ}_2$ . To find, on this supposition, the vector described in $dt$, we must double $\overline{PQ}_2$ , and we find, as a second approximation to the vector which the moving point would have described in time $dt$, if it had moved for that period in the direction and with the velocity it had at $P$,

$$\begin{aligned} \overline{Pq}_2 = 2\overline{PQ}_2 &= 2(\overline{OQ}_2 - \overline{OP}) \\ &= 2\{\phi(t + \tfrac{1}{2}dt) - \phi(t)\} \end{aligned}$$

The next approximation gives

$$\begin{aligned} \overline{Pq}_3 = 3\overline{PQ}_3 &= 3(\overline{OQ}_3 - \overline{OP}) \\ &= 3\{\phi(t + \tfrac{1}{3}dt) - \phi(t)\} \end{aligned}$$

And so on, each step evidently leading us nearer the sought truth. Hence, to find the vector which would have been described in time $dt$ had the circumstances of the motion at $P$ remained undisturbed, we must find the value of

$$d\rho = \overline{Pq} = L_{x=\infty} x \left\{ \phi\left(t + \frac{1}{x}dt\right) - \phi(t) \right\}$$

We have seen that in this particular case we may use Taylor's Theorem. We have, therefore,

$$\begin{aligned} d\rho &= L_{x=\infty} x \left\{ \phi'(t) \tfrac{1}{x} dt + \phi''(t) \tfrac{1}{x^2} \tfrac{(dt)^2}{1 \,.\, 2} + \&\text{c} \right\} \\ &= \phi'(t) dt \end{aligned}$$

And, if we choose, we may now write

$$\frac{d\rho}{dt} = \phi'(t)$$

**37**. But it is to be most particularly remarked that in the whole of this investigation no regard whatever has been paid to the magnitude of $dt$. The question which we have now answered may be put in the form – *A point describes a given curve in a given manner. At any point of its path its motion suddenly ceases to be accelerated. What space will it describe in a definite interval?* As Hamilton well observes, this is, for a planet or comet, the case of a 'celestial Atwood's machine'.

**38**. If we suppose the variable, in terms of which $\rho$ is expressed, to be the arc, $s$, of the curve measured from some fixed point, we find as before

$$d\rho = \phi'(x) ds$$

From the very nature of the question it is obvious that the length of $dp$ must in this case be $ds$, so that $\phi'(s)$ is necessarily a unit-vector. This remark is of importance, as we will see later; and it may therefore be useful to obtain afresh the above result without any reference to time or velocity.

**39**. Following strictly the process of Newton s VIIth Lemma, let us describe on $Pq_2$ an arc similar to $PQ_2$, and so on. Then obviously, as the subdivision of $ds$ is carried farther, the new arc (whose length is always $ds$) more and more nearly (and without limit) coincides with the line which expresses the corresponding approximation to $dp$.

**40**. As additional examples let us take some well-known *plane* curves; and first the hyperbola (§31 ($k$))

$$\rho = \alpha t + \frac{\beta}{t}$$

Here

$$d\rho = \left( \alpha - \frac{\beta}{t^2} \right) dt$$

This shows that the tangent is parallel to the vector

$$\alpha t - \frac{\beta}{t}$$

In words, *if the vector (from the centre) of a point in a hyperbola be one diagonal of a parallelogram, two of whose sides coincide with the asymptotes, the other diagonal is parallel to the tangent at the point, and cuts off a constant area from the space between the asymptotes.* (For the sides of this triangular area are $t$ times the length of $\alpha$, and $1/t$ times the length of $\beta$, respectively; the angle between them being constant.)

Next, take the cycloid, as in §31 ($k$),

$$\rho = \alpha(t + \sin t) + \beta \cos t$$

We have

$$d\rho = \{\alpha(1+\cos t) - \beta \sin t\}dt$$

At the vertex

$$t = 0, \qquad \cos t = 1, \qquad \sin t = 0, \qquad \text{and } d\rho = 2\alpha dt$$

At a cusp

$$t = \pi, \qquad \cos t = -1, \qquad \sin t = 0, \qquad \text{and } d\rho = 0$$

This indicates that, at the cusp, the tracing point is ( instantaneously) at rest. To find the direction of the tangent, and the form of the curve in the vicinity of the cusp, put $t = \pi + \tau$, where powers of $\tau$ above the second are omitted. We have

$$d\rho = \beta\tau dt + \frac{\alpha\tau^2}{2}dt$$

so that, at the cusp, the tangent is parallel to $\beta$. By making the same substitution in the expression for $\rho$, we find that the part of the curve near the cusp is a semicubical parabola,

$$\rho = \alpha(\pi + \tau^3/6) - \beta(1 - \tau^2/2)$$

or, if the origin be shifted to the cusp ($\rho = \pi\alpha - \beta$),

$$\rho = \alpha\tau^3/6 + \beta\tau^2/2$$

**41**. Let us reverse the first of these questions, and *seek the envelope of a line which cuts off from two fixed axes a triangle of constant area.*

If the axes be in the directions of $\alpha$ and $\beta$, the intercepts may evidently be written $\alpha t$ and $\frac{\beta}{t}$. Hence the equation of the line is (§30)

$$\rho = \alpha t + x\left(\frac{\beta}{t} - \alpha t\right)$$

The condition of envelopment is, obviously, (see Chap. IX.)

$$d\rho = 0$$

This gives $0 = \left\{\alpha - x\left(\frac{\beta}{t^2} + \alpha\right)\right\}dt + \left(\frac{\beta}{t} - \alpha t\right)dx$ [2]

Hence $$(1-x)dt - tdx = 0$$

and $$-\frac{x}{t^2}dt + \frac{dx}{t} = 0$$

From these, at once, $x = \frac{1}{2}$, since $dx$ and $dt$ are indeterminate. Thus the equation of the envelope is

$$\begin{aligned} \rho &= \alpha t + \tfrac{1}{2}\left(\tfrac{\beta}{t} - \alpha t\right) \\ &= \tfrac{1}{2}\left(\alpha t + \tfrac{\beta}{t}\right) \end{aligned}$$

[2] Here we have opportunity for a remark (very simple indeed, but) of the utmost importance. *We are not to equate separately to zero the coefficients of dt and dx*; for we must remember that this equation is of the form

$$0 = p\alpha + q\beta$$

where $p$ and $q$ are numbers; and that, so long as $\alpha$ and $\beta$ are actual and non-parallel vectors, the existence of such an equation requires (§24)

the hyperbola as before; $\alpha$, $\beta$ being portions of its asymptotes.

**42**. It may assist the student to a thorough comprehension of the above process, if we put it in a slightly different form. Thus the equation of the enveloping line may be written

$$\rho = \alpha t(1-x) + \beta\frac{x}{t}$$

which gives

$$d\rho = 0 = \alpha d\{t(1-x)\} + \beta d\left(\frac{x}{t}\right)$$

Hence, as $\alpha$ is not parallel to $\beta$, we must have

$$d\{t(1-x)\} = 0, \qquad d\left(\frac{x}{t}\right) = 0$$

and these are, when expanded, the equations we obtained in the preceding section.

**43**. For farther illustration we give a solution not directly employing the differential calculus. The equations of any two of the enveloping lines are

$$\rho = \alpha t + x\left(\frac{\beta}{t} - \alpha t\right)$$

$$\rho = \alpha t_1 + x_1\left(\frac{\beta}{t_1} - \alpha t_1\right)$$

$t$ and $t_1$ being given, while $x$ and $x_1$ are indeterminate.

At the point of intersection of these lines we have (§26),

$$\left.\begin{array}{rcl} t(1-x) & = & t_1(1-x_1) \\ \frac{x}{t} & = & \frac{x_1}{t_1} \end{array}\right\}$$

These give, by eliminating $x_1$

$$t(1-x) = t_1\left(1 - \frac{t_1}{t}x\right)$$

or $$x = \frac{t}{t_1+t}$$

Hence the vector of the point of intersection is

$$\rho = \frac{\alpha t t_1 + \beta}{t_1 + t}$$

and thus, for the ultimate intersections, where $L\frac{t_1}{t} = 1$,

$$\rho = \frac{1}{2}\left(\alpha t + \frac{\beta}{t}\right) \text{ as before}$$

COR. If. instead of the *ultimate* intersections, we consider the intersections of pairs of these lines related by some law, we obtain useful results. Thus let

$$tt_1 = 1$$

$$\rho = \frac{\alpha + \beta}{t + \frac{1}{t}}$$

or the intersection lies in the diagonal of the parallelogram on $\alpha$, $\beta$.

If $t_1 = mt$, where $m$ is constant,

$$\rho = \frac{mt\alpha + \frac{\beta}{t}}{m + 1}$$

But we have also $x = \frac{1}{m+1}$

Hence *the locus of a point which divides in a given ratio a line cutting off a given area from two fixed axes, is a hyperbola of which these axes are the asymptotes.*

If we take either

$$tt_1(t + t_1) = \text{constant, or } \frac{t^2 t_1^2}{t + t_1} = \text{constant}$$

the locus is a parabola; and so on.

It will be excellent practice for the student, at this stage, to work out in detail a number of similar questions relating to the envelope of, or the locus of the intersection of selected pairs from, a series of lines drawn according to a given law. And the process may easily be extended to planes. Thus, for instance, we may form the general equation of planes which cut off constant tetrahedra from the axes of coordinates. Their envelope is a surface of the third degree whose equation may be written

$$\rho = x\alpha + y\beta + z\gamma$$

where $$xyz = \alpha^3$$

Again, find the locus of the point of intersection of three of this group of planes, such that the first intercepts on $\beta$ and $\gamma$, the second on $\gamma$ and $\alpha$, the third on $\alpha$ and $\beta$, lengths all equal to one another, &c. But we must not loiter with such simple matters as these.

**44**. The reader who is fond of Anharmonic Ratios and Trans versals will find in the early chapters of Hamilton's *Elements of Quaternions* an admirable application of the composition of vectors to these subjects. The Theory of Geometrical Nets, in a plane, and in space, is there very fully developed; and the method is shown to include, as particular cases, the corresponding processes of Grassmann's *Ausdehnungslehre* and Möbius' *Barycentrische Calcul*. Some very curious investigations connected with curves and surfaces of the second and third degrees are also there founded upon the composition of vectors.

## 4.3 Examples To Chapter 1.

1. The lines which join, towards the same parts, the extremities of two equal and parallel lines are themselves equal and parallel. (*Euclid*, I. xxxiii.)

2. Find the vector of the middle point of the line which joins the middle points of the diagonals of any quadrilateral, plane or gauche, the vectors of the corners being given; and so prove that this point is the mean point of the quadrilateral.

If two opposite sides be divided proportionally, and two new quadrilaterals be formed by joining the points of division, the mean points of the three quadrilaterals lie in a straight line.

Show that the mean point may also be found by bisecting the line joining the middle points of a pair of opposite sides.

3. Verify that the property of the coefficients of three vectors whose extremities are in a line (§30) is not interfered with by altering the origin.

4. If two triangles $ABC$, $abc$, be so situated in space that $Aa$, $Bb$, $Cc$ meet in a point, the intersections of $AB$, $ab$, of $BC$, $bc$, and of $CA$, $ca$, lie in a straight line.

5. Prove the converse of 4, i.e. if lines be drawn, one in each of two planes, from any three points in the straight line in which these planes meet, the two triangles thus formed are sections of a common pyramid.

6. If five quadrilaterals be formed by omitting in succession each of the sides of any pentagon, the lines bisecting the diagonals of these quadrilaterals meet in a point. (H. Fox Talbot.)

7. Assuming, as in §7, that the operator

$$\cos\theta + \sqrt{-1}\sin\theta$$

turns any radius of a given circle through an angle $\theta$ in the positive direction of rotation, without altering its length, deduce the ordinary formulae for $\cos(A+B)$, $\cos(A-B)$, $\sin(A+B)$, and $\sin(A-B)$, in terms of sines and cosines of $A$ and $B$.

8. If two tangents be drawn to a hyperbola, the line joining the centre with their point of intersection bisects the lines join ing the points where the tangents meet the asymptotes : and the secant through the points of contact bisects the intercepts on the asymptotes.

9. Any two tangents, limited by the asymptotes, divide each other proportionally.

10. If a chord of a hyperbola be one diagonal of a parallelogram whose sides are parallel to the asymptotes, the other diagonal passes through the centre.

11. Given two points $A$ and $B$, and a plane, $C$. Find the locus of $P$, such that if $AP$ cut $C$ in $Q$, and $BP$ cut $C$ in $R$, $\overline{QR}$ may be a given vector.

12. Show that $\rho = x^2\alpha + y^2\beta + (x+y)^2\gamma$
is the equation of a cone of the second degree, and that its section by the plane

$$\rho = \frac{p\alpha + q\beta + r\gamma}{p+q+r}$$

is an ellipse which touches, at their middle points, the sides of the triangle of whose corners $\alpha$, $\beta$, $\gamma$ are the vectors. (Hamilton, *Elements*, p. 96.)

13. The lines which divide, proportionally, the pairs of opposite sides of a gauche quadrilateral, are the generating lines of a hyperbolic paraboloid. (*Ibid.* p. 97.)

14. Show that $\rho = x^3\alpha + y^3\beta + z^3\gamma$
where $x+y+z=0$
represents a cone of the third order, and that its section by the plane

$$\rho = \frac{p\alpha + q\beta + r\gamma}{p+q+r}$$

is a cubic curve, of which the lines

$$\rho = \frac{p\alpha + q\beta}{p+q}, \text{ \&c}$$

are the asymptotes and the three (real) tangents of inflection. Also that the mean point of the triangle formed by these lines is a conjugate point of the curve. Hence that the vector $\alpha + \beta + \gamma$ is a conjugate ray of the cone. (*Ibid.* p. 96.)

## 4.4 Products And Quotients of Vectors

**45**. We now come to the consideration of questions in which the Calculus of Quaternions differs entirely from any previous mathematical method; and here we will get an idea of what a Quaternion is, and whence it derives its name. These questions are fundamentally involved in the novel use of the symbols of multiplication and division. And the simplest introduction to the subject seems to be the consideration of the quotient, or ratio, of two vectors.

**46**. If the given vectors be parallel to each other, we have already seen (§22) that either may be expressed as a numerical multiple of the other; the multiplier being simply the ratio of their lengths, taken positively if they have similar currency, negatively if they run opposite ways.

**47**. If they be not parallel, let $\overline{OA}$ and $\overline{OB}$ be drawn parallel and equal to them from any point $O$; and the question is reduced to finding the value of the ratio of two vectors drawn from the same point. Let us first find *upon how many distinct numbers this ratio depends*.

We may suppose $\overline{OA}$ to be changed into $\overline{OB}$ by the following successive processes.

1st. Increase or diminish the length of $\overline{OA}$ till it becomes equal to that of $\overline{OB}$. For this only one number is required, viz. the ratio of the lengths of the two vectors. As Hamilton remarks, this is a positive, or rather a *signless*, number.

2nd. Turn $\overline{OA}$ about $O$, in the common plane of the two vectors, until its direction coincides with that of $\overline{OB}$, and (remembering the effect of the first operation) we see that the two vectors now coincide or become identical. To specify this operation three numbers are required, viz. two angles (such as node and inclination in the case of a planet's orbit) to fix the plane in which the rotation takes place, and *one* angle for the amount of this rotation.

Thus it appears that the ratio of two vectors, or the multiplier required to change one vector into another, in general depends upon *four* distinct numbers, whence the name QUATERNION.

A quaternion q is thus *defined* as expressing a relation

$$\beta = q\alpha$$

between two vectors $\alpha$, $\beta$. By what precedes, the vectors $\alpha$, $\beta$, which serve for the definition of a given quaternion, must be in a given plane, at a given inclination to each other, and with their lengths in a given ratio ; but it is to be noticed that they may be *any* two such vectors. [*Inclination* is understood to include sense, or currency, of rotation from $\alpha$ to $\beta$.]

The particular case of perpendicularity of the two vectors, where their quotient is a vector perpendicular to their plane, is fully considered below; §§64, 65, 72, &c.

**48**. It is obvious that the operations just described may be performed, with the same result, in the opposite order, being perfectly independent of each other. Thus it appears that a quaternion, considered as the factor or agent which changes one definite vector into another, may itself be decomposed into two factors of which the order is immaterial.

The *stretching* factor, or that which performs the first operation in §47, is called the TENSOR, and is denoted by prefixing $T$ to the quaternion considered.

The *turning factor*, or that corresponding to the second operation in §47, is called the VERSOR, and is denoted by the letter $U$ prefixed to the quaternion.

**49**. Thus, if $\overline{OA} = \alpha$, $\overline{OB} = \beta$, and if $q$ be the quaternion which changes $\alpha$ to $\beta$, we have

$$\beta = q\alpha$$

which we may write in the form

$$\frac{\beta}{\alpha} = q, \text{ or } \beta\alpha^{-1} = q$$

if we agree to *define* that

$$\frac{\beta}{\alpha}\alpha = \beta\alpha^{-1}\alpha = \beta$$

Here it is to be particularly noticed that we write $q$ *before* $\alpha$ to signify that $\alpha$ is multiplied by (or operated on by) $q$, not $q$ multiplied by $\alpha$.

This remark is of extreme importance in quaternions, for, as we will soon see, the Commutative Law does not generally apply to the factors of a product.

We have also, by §§47, 48,

$$q = TqUq = UqTq$$

where, as before, $Tq$ depends merely on the relative lengths of $\alpha$ and $\beta$, and $Uq$ depends solely on their directions.

Thus, if $\alpha_1$ and $\beta_1$ be vectors of unit length parallel to $\alpha$ and $\beta$ respectively,

$$T\frac{\beta_1}{\alpha_1} = T\beta_1/T\alpha_1 = 1,\ U\frac{\beta_1}{\alpha_1} = U\beta_1/U\alpha_1 = U\frac{\beta}{\alpha}$$

As will soon be shown, when $\alpha$ is perpendicular to $\beta$, i.e. when the versor of the quotient is quadrantal, it is a unit-vector.

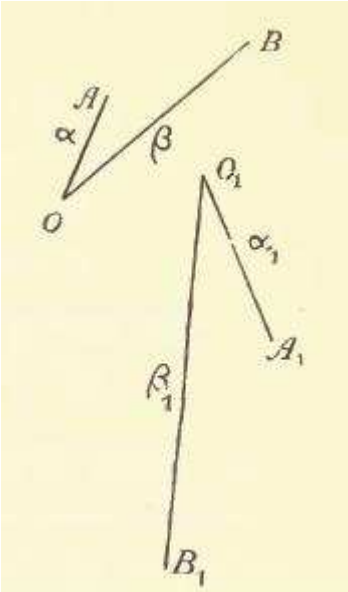
**50**. We must now carefully notice that the quaternion which is the quotient when $\beta$ is divided by $\alpha$ in no way depends upon the *absolute* lengths, or directions, of these vectors. Its value will remain unchanged if we substitute for them any other pair of vectors which

(1) have their lengths in the same ratio,

(2) have their common plane the same or parallel,

and (3) make the same angle with each other.

Thus in the annexed figure




$$\frac{O_1B_1}{O_1A_1} = \frac{\overline{OB}}{\overline{OA}}$$

if, and only if,

(1) $\frac{O_1B_1}{O_1A_1} = \frac{OB}{OA}$

(2) plane $AOB$ parallel to plane $A_1O_1B_1$

(3) $\angle AOB = \angle A_1O_1B_1$

[Equality of angles is understood to include concurrency of rotation. Thus in the annexed figure the rotation about an axis drawn upwards from the plane is negative (or clock- wise) from $OA$ to $OB$, and also from $O_1A_1$ to $O_1B_1$.]

It thus appears that if

$$\beta = q\alpha,\ \delta = q\gamma$$

the vectors $\alpha$, $\beta$, $\gamma$, $\delta$ are parallel to one plane, and may be repre sented (in a highly extended sense) as *proportional* to one another, thus: –

$$\beta : \alpha = \delta : \gamma$$

And it is clear from the previous part of this section that this may be written not only in the form

$$\alpha : \beta = \gamma : \delta$$

but also in either of the following forms: –

$$\gamma : \alpha = \delta : \beta$$

$$\alpha : \gamma = \beta : \delta$$

While these proportions are true as equalities of ratios, they do not usually imply equalities of products.

Thus, as the first of these was equivalent to the equation

$$\frac{\beta}{\alpha} = \frac{\delta}{\gamma} = q, \text{ or } \beta\alpha^{-1} = \delta\gamma^{-1} = q$$

the following three imply separately, (see next section)

$$\frac{\alpha}{\beta} = \frac{\gamma}{\delta} = q^{-1},\ \frac{\gamma}{\alpha} = \frac{\delta}{\beta} = r,\ \frac{\alpha}{\gamma} = \frac{\beta}{\delta} = r^{-1}$$

or, if we please,

$$\alpha\beta^{-1} = \gamma\delta^{-1} = q^{-1},\ \gamma\alpha^{-1} = \delta\beta^{-1} = r,\ \alpha\gamma^{-1} = \beta\delta^{-1} = r^{-1}$$

where $r$ is a *new* quaternion, which has not necessarily anything (except its plane), in common with $q$.

But here great caution is requisite, for we are *not* entitled to conclude from these that

$$\alpha\delta = \beta\gamma, \text{ \&c.}$$

This point will be fully discussed at a later stage. Meanwhile we may merely *state* that from

$$\frac{\alpha}{\beta} = \frac{\gamma}{\delta}, \text{ or } \frac{\beta}{\alpha} = \frac{\delta}{\gamma}$$

we are entitled to deduce a number of equivalents such as

$$\alpha\beta^{-1}\delta = \gamma, \text{ or } \alpha = \gamma\delta^{-1}\beta, \text{ or } \beta^{-1}\delta = \alpha^{-1}\gamma, \text{ \&c}$$

**51**. The *Reciprocal* of a quaternion $q$ is defined by the equation

$$\frac{1}{q}q = q^{-1} = 1 = q\frac{1}{q} = qqe^{-1}$$

Hence if

$$\frac{\beta}{\alpha} = q, \text{ or}$$
$$\beta = q\alpha$$

we must have

$$\frac{\alpha}{\beta} = \frac{1}{q} = q^{-1}$$

For this gives

$$\frac{\alpha}{\beta}\beta = q^{-1}q\alpha$$

and each member of the equation is evidently equal to $\alpha$. Or thus: –

$$\beta = q\alpha$$

Operate *by* $q^{-1}$

$$q^{-1}\beta = \alpha$$

Operate *on* $\beta^{-1}$

$$q^{-1} = \alpha\beta^{-1} = \frac{\alpha}{\beta}$$

Or, we may reason thus: – since $q$ changes $\overline{OA}$ to $\overline{OA}$, $q^{-1}$ must change $\overline{OB}$ to $\overline{OA}$, and is therefore expressed by $\frac{\alpha}{\beta}$ (§49).

The tensor of the reciprocal of a quaternion is therefore the reciprocal of the tensor; and the versor differs merely by the *reversal* of its representative angle. The versor, it must be remembered, gives the plane and angle of the turning – it has nothing to do with the extension.

[*Remark.* In §§49–51, above, we had such expressions as $\frac{\beta}{\alpha} = \beta\alpha^{-1}$. We have also met with $\alpha^{-1}\beta$. Cayley suggests that this also may be written in the ordinary fractional form by employing the following distinctive notation: –

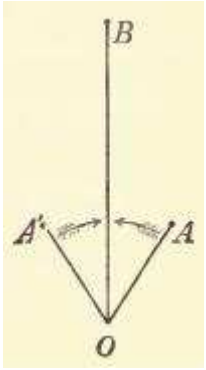
$$\frac{\beta}{\alpha} = \beta\alpha^{-1} = \frac{\beta|}{|\alpha}, \; \alpha^{-1}\beta = \frac{|\beta}{\alpha|}$$

(It might, perhaps, be even simpler to use the *solidus* as recommended by Stokes, along with an obviously correlative type:– thus,

$$\frac{\beta}{\alpha} = \beta\alpha^{-1} = \beta/\alpha, \; \alpha^{-1}\beta = \alpha\beta$$

I have found such notations occasionally convenient for private work, but I hesitate to introduce changes unless they are abso lutely required. See remarks on this point towards the end of the *Preface to the Second Edition* reprinted above.]

**52**. The *Conjugate* of a quaternion $q$, written $Kq$, has the same tensor, plane, and angle, only the angle is taken the reverse way; or the versor of the conjugate is the reciprocal of the versor of the quaternion, or (what comes to the same thing) the versor of the reciprocal.




Thus, if $OA$, $OB$, $OA'$ , lie in one plane, and if $OA' = OA$, and $\angle A'OB = \angle BOA$, we have

$$\frac{\overline{OB}}{\overline{OA}} = q$$

, and

$$\frac{\overline{OB}}{\overline{OA'}} = \text{ congugate of } q = Kq$$

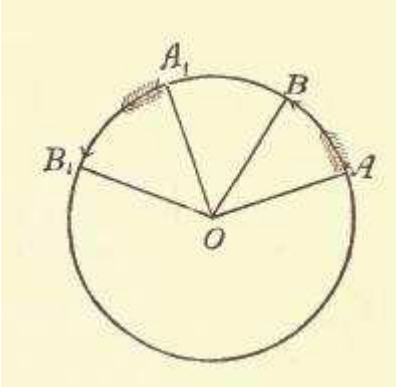
By last section we see that

$$Kq = (Tq)^2 q^{-1}$$

Hence $$qKq = Kqq = (Tq)^2$$

This proposition is obvious, if we recollect that the tensors of $q$ and $Kq$ are equal, and that the versors are such that either *annuls* the effect of the other; while the order of their application is indifferent. The joint effect of these factors is therefore merely to multiply twice over by the common tensor.

**53**. It is evident from the results of §50 that, if $\alpha$ and $\beta$ be of equal length, they may be treated as of unit-length so far as their quaternion quotient is concerned. This quotient is therefore a versor (the tensor being unity) and may be represented indifferently by any one of an infinite number of concurrent arcs of given length lying on the circumference of a circle, of which the two vectors are radii. This is of considerable importance in the proofs which follow.



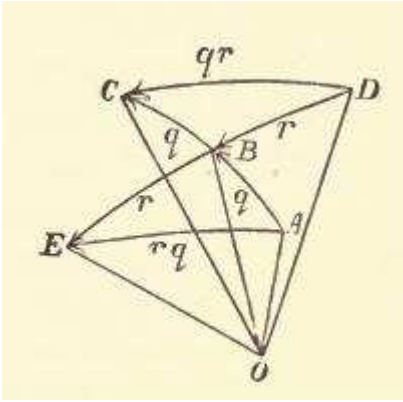

Thus the versor $\dfrac{\overline{OB}}{\overline{OA}}$ may be represented in magnitude, plane, and currency of rotation (§50) by the arc $AB$, which may in this extended sense be written $\overset{\frown}{AB}$.

And, similarly, the versor $\dfrac{\overline{OB_1}}{\overline{OA_1}}$ may be represented by $\overset{\frown}{A_1B_1}$ which is equal to (and concurrent with) $\overset{\frown}{AB}$ if

$$\angle A_1OB_1 = \angle AOB$$

i.e. if the versors are *equal*, in the quaternion meaning of the word.

**54**. By the aid of this process, when a versor is represented as an arc of a great circle on the unit-sphere, we can easily prove that *quaternion multiplication is not generally commutative*.




Thus let $q$ be the versor $\overset{\frown}{AB}$ or $\dfrac{\overline{OB}}{\overline{OA}}$, where $O$ is the centre of the sphere.

Take $\overset{\frown}{BC} = \overset{\frown}{AB}$, (which, it must be remembered, makes the points $A$, $B$, $C$, lie in one great circle), then $q$ may also be represented by $\dfrac{\overline{OC}}{\overline{OB}}$.

In the same way any other versor $r$ may be represented by $\overset{\frown}{DB}$ or $\overset{\frown}{BE}$ and by $\dfrac{\overline{OB}}{\overline{OD}}$ or $\dfrac{\overline{OE}}{\overline{OB}}$.

[The line $OB$ in the figure is definite, and is given by the intersection of the planes of the two versors.]

Now $r\overline{OD} = \overline{OB}$, and $q\overline{OB} = \overline{OC}$.

Hence $qr\overline{OD} = \overline{OC}$,

or $qr = \dfrac{\overline{OC}}{\overline{OD}}$, and may therefore be represented by the arc $\overset{\frown}{DC}$ of a great circle.

But $rq$ is easily seen to be represented by the arc $\overset{\frown}{AE}$.

For $q\overline{OA} = \overline{OB}$, and $r\overline{OB} = \overline{OE}$,

whence $rq\overline{OA} = \overline{OE}$. and $rq = \dfrac{\overline{OE}}{\overline{OA}}$.

Thus the versors $rq$ and $qr$, though represented by arcs of equal length, are not generally in the same plane and are therefore unequal: unless the planes of $q$ and $r$ coincide.

Remark. We see that we have assumed, or defined, in the above proof, that $q.r\alpha = qr.\alpha$. and $r.q\alpha = rq.\alpha$ in the special case when $q\alpha$, $r\alpha$, $q.r\alpha$ and $r.q\alpha$ are all *vectors*.

**55**. Obviously $\overset{\frown}{CB}$ is $Kq$, $\overset{\frown}{BD}$ is $Kr$, and $\overset{\frown}{CD}$ is $K(qr)$. But $\overset{\frown}{CD} = \overset{\frown}{BD}.\overset{\frown}{CB}$ as we see by applying both to OC. This gives us the very important theorem

$$K(qr) = Kr.Kq$$

i.e. *the conjugate of the product of two versors is the product of their conjugates in inverted order*. This will, of course, be extended to any number of factors as soon as we have proved the associative property of multiplication. (§58 below.)

**56**. The propositions just proved are, of course, true of quater nions as well as of versors; for the former involve only an additional numerical factor which has reference to the length

merely, and not the direction, of a vector (§48), and is therefore commutative with all other factors.

**57**. Seeing thus that the commutative law does not in general hold in the multiplication of quaternions, let us enquire whether the Associative Law holds generally. That is if $p$, $q$, $r$ be three quaternions, have we

$$p.qr = pq.r?$$

This is, of course, obviously true if $p$, $q$, $r$ be numerical quantities, or even any of the imaginaries of algebra. But it cannot be con sidered as a truism for symbols which do not in general give

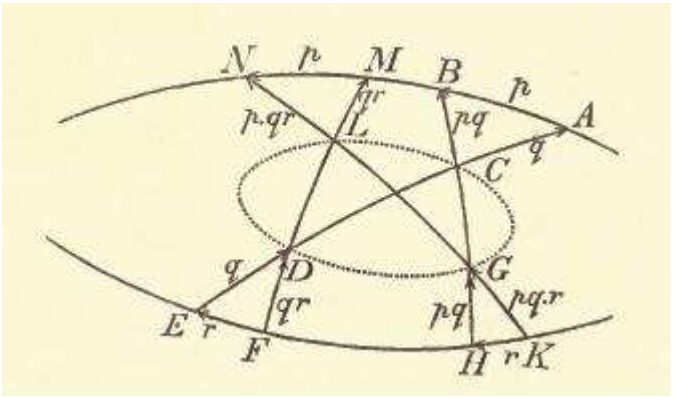
$$pq = qp$$

We have assumed it, in definition, for the special case when $r$, $qr$, and $pqr$ are all vectors. (§54.) But we are not entitled to assume any more than is absolutely required to make our definitions complete.

**58**. In the first place we remark that $p$, $q$, and $r$ may be considered as versors only, and therefore represented by arcs of great circles on the unit sphere, for their tensors may obviously (§48) be divided out from both sides, being commutative with the versors.

Let $\overset{\frown}{AB} = p$, $\overset{\frown}{ED} = \overset{\frown}{CA} = q$, and $\overset{\frown}{FE} = r$.

Join $BC$ and produce the great circle till it meets $EF$ in $H$, and make $\overset{\frown}{KH} = \overset{\frown}{FE} = r$, and $\overset{\frown}{HG} = \overset{\frown}{CB} = pq$ (§54).




Join $GK$. Then $\overset{\frown}{KG} = \overset{\frown}{HG}.\overset{\frown}{KH} = pq.r$.

Join $FD$ and produce it to meet $AB$ in $M$. Make

$$\overset{\frown}{LM} = \overset{\frown}{FD}, \text{ and } \overset{\frown}{MN} = \overset{\frown}{AB}$$

and join $NL$. Then

$$\overset{\frown}{LN} = \overset{\frown}{MN}.\overset{\frown}{LM} = p.qr$$

.

Hence to show that $p.qr = pq.r$

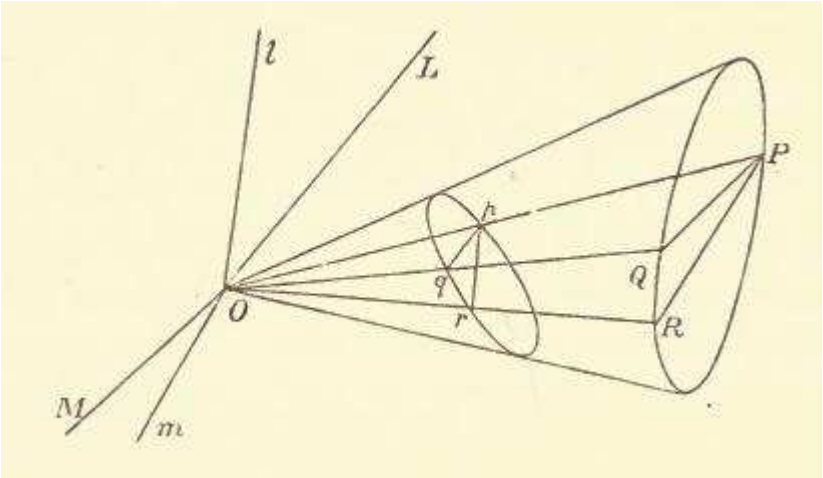
all that is requisite is to prove that $LN$, and $KG$, described as above, are *equal arcs of the same great circle*, since, by the figure, they have evidently similar currency. This is perhaps most easily effected by the help of the fundamental properties of the curves known as *Spherical Conics*. As they are not usually familiar to students, we make a slight digression for the purpose of proving these fundamental properties ; after Chasles, by whom and Magnus they were discovered. An independent proof of the associative principle will presently be indicated, and in Chapter VIII. we will employ quaternions to give an independent proof of the theorems now to be established.

**59**.* DEF. *A spherical conic is the curve of intersection of a cone of the second degree with a sphere, the vertex of the cone being the centre of the sphere.*

LEMMA. If a cone have one series of circular sections, it has another series, and any two circles belonging to different series lie on a sphere. This is easily proved as follows.

Describe a sphere, $A$, cutting the cone in one circular section, $C$, and in any other point whatever, and let the side $OpP$ of the cone meet $A$ in $p$, $P$ ; $P$ being a point in $C$. Then $PO.Op$ is constant, and, therefore, since $P$ lies in a plane, $p$ lies on a sphere, $a$, passing through 0. Hence the locus, $c$, of $p$ is a circle, being the intersection of the two spheres $A$ and $a$.

Let $OqQ$ be any other side of the cone, $q$ and $Q$ being points in $c$, $C$ respectively. Then the quadrilateral $qQPp$ is inscribed in a circle (that in which its plane cuts the sphere $A$) and the exterior




angle at $p$ is equal to the interior angle at $Q$. If $OL$, $OM$ be the lines in which the plane $POQ$ cuts the *cyclic planes* (planes through $O$ parallel to the two series of circular sections) they are obviously parallel to $pq$, $QP$, respectively; and therefore

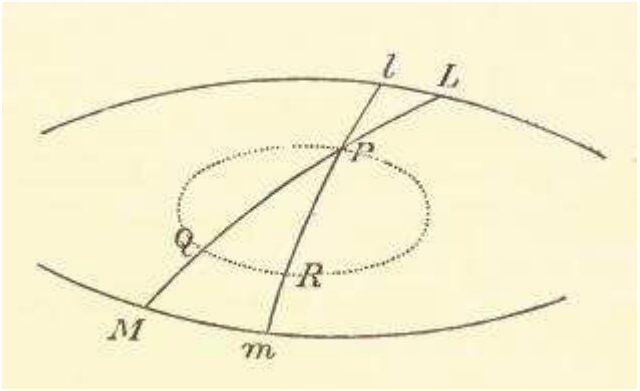
$$\angle LOp = \angle Opq = \angle OQP = \angle MOQ$$

Let any third side, $OrR$, of the cone be drawn, and let the plane $OPR$ cut the cyclic planes in $0l$, $Om$ respectively. Then, evidently,

$$\angle lOL = \angle qpr$$

$$\angle MOm = \angle QPR$$

and these angles are independent of the position of the points $p$ and $P$, if $Q$ and $R$ be fixed points.




In the annexed section of the above space-diagram by a sphere whose centre is $O$, $lL$, $Mm$ are the great circles which represent the cyclic planes, $PQR$ is the spherical conic which represents the cone. The point $P$ represents the line $OpP$, and so with the others. The propositions above may now be stated thus,

$$\text{Arc } PL = \text{arc } MQ$$

and, if $Q$ and $R$ be fixed, $Mm$ and $lL$ are constant arcs whatever be the position of $P$.

**60**. The application to §58 is now obvious. In the figure of that article we have

$$\overset{\frown}{FE} = \overset{\frown}{KH},\ \overset{\frown}{ED} = \overset{\frown}{CA},\ \overset{\frown}{HG} = \overset{\frown}{CB},\ \overset{\frown}{LM} = \overset{\frown}{FD}$$

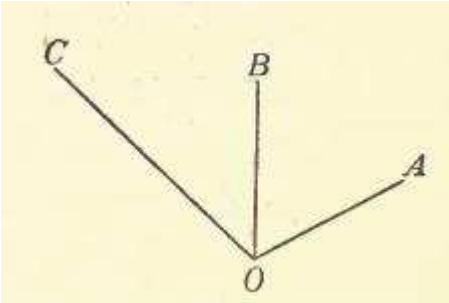
Hence $L$, $C$, $G$, $D$ are points of a spherical conic whose cyclic planes are those of $AB$, $FE$. Hence also $KG$ passes through $L$, and with $LM$ intercepts on $AB$ an arc equal to $\overset{\frown}{AB}$. That is, it passes through $N$, or $KG$ and $LN$ are arcs of the same great circle : and they are equal, for $G$ and $L$ are points in the spherical conic.

Also, the associative principle holds for any number of quaternion factors. For, obviously,

$$qr.st = qrs.t = \&\text{c.}, \&\text{c.},$$

since we may consider $qr$ as a single quaternion, and the above proof applies directly.

**61**. That quaternion addition, and therefore also subtraction, is commutative, it is easy to show.




For if the planes of two quaternions, $q$ and $r$, intersect in the line $OA$, we may take any vector $\overline{OA}$ in that line, and at once find two others, $\overline{OB}$ and $\overline{OC}$, such that

$$\overline{OB} = q\overline{OA}$$

and $$\overline{CO} = r\overline{OA}$$

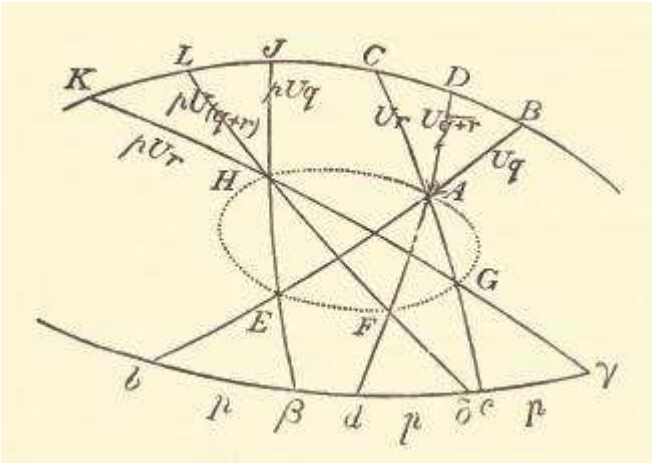
And $$(q+r)\overline{OA}\overline{OB} + \overline{OC} = \overline{OC} + \overline{OB} = (r+q)\overline{OA}$$

since vector addition is commutative (§27).

Here it is obvious that $(q+r)\overline{OA}$, being the diagonal of the parallelogram on $\overline{OB}$, $\overline{OC}$, divides the angle between $OB$ and $OC$ in a ratio depending solely on the ratio of the lengths of these lines, i.e. on the ratio of the tensors of $q$ and $r$. This will be useful to us in the proof of the distributive law, to which we proceed.

**62**. Quaternion multiplication, and therefore division, is distributive. One simple proof of this depends on the possibility, shortly to be proved, of representing *any* quaternion as a linear function of three given rectangular unit- vectors. And when the proposition is thus established, the associative principle may readily be deduced from it.

[But Hamilton seems not to have noticed that we may employ for its proof the properties of Spherical Conies already employed




in demonstrating the truth of the associative principle. "For continuity we give an outline of the proof by this process.

Let $\overset{\frown}{BA}$, $\overset{\frown}{CA}$ represent the versors of $q$ and $r$, and be the great circle whose plane is that of $p$.

Then, if we take as operand the vector $\overline{OA}$, it is obvious that $U(q+r)$ will be represented by some such arc as $\overset{\frown}{DA}$ where $B$, $D$, $C$ are in one great circle; for $(q+r)\overline{OA}$ is in the same plane as $q\overline{OA}$ and $r\overline{OA}$, and the relative magnitude of the arcs $BD$ and $DC$ depends solely on the tensors of $q$ and $r$. Produce $BA$, $DA$, $CA$ to meet be in $b$, $d$, $c$ respectively, and make

$$\overset{\frown}{Eb} = \overset{\frown}{BA}, \overset{\frown}{Fd} = \overset{\frown}{DA}, \overset{\frown}{Gc} = \overset{\frown}{CA}$$

Also make $\overset{\frown}{b\beta} = \overset{\frown}{d\delta} = \overset{\frown}{c\gamma} = p$. Then $E$, $F$, $G$, $A$ lie on a spherical conic of which $BC$ and $bc$ are the cyclic arcs. And, because $\overset{\frown}{b\beta} = \overset{\frown}{d\delta} = \overset{\frown}{c\gamma}$, $\overset{\frown}{\beta E}$, $\overset{\frown}{\delta F}$, $\overset{\frown}{\gamma G}$, when produced, meet in a point $H$ which is also on the spherical conic (§59*). Let these arcs meet $BC$ in $J$, $L$, $K$ respectively. Then we have

$$\overset{\frown}{JH} = \overset{\frown}{E\beta} = pUq$$

$$\overset{\frown}{LH} = \overset{\frown}{F\delta} = pU(q+r)$$

$$\overset{\frown}{KH} = \overset{\frown}{G\gamma} = pUr$$

Also $\overset{\frown}{LJ} = \overset{\frown}{DB}$

and $\overset{\frown}{KL} = \overset{\frown}{CD}$

And, on comparing the portions of the figure bounded respectively by $HKJ$ and by $ACB$ we see that (when considered with reference to their effects as factors multiplying $\overline{OH}$ and $\overline{OA}$ respectively)

$pU(q4+r)$ bears the same relation to $pUq$ and $pUr$
that $U(q+r)$ bears to $Uq$ and $Ur$.
But $T(q+r)U(q+r) = q+r = TqUq + TrUr$.
Hence $T(q+r).pU(q+r) = Tq.pUq + Tr.pUr$;
or, since the tensors are mere numbers and commutative with all other factors,

$$p(q+r) = pq + pr$$

In a similar manner it may be proved that

$$(q+)p = qp + rp$$

And then it follows at once that

$$(p+q)(r+s) = pr + ps + qr + qs$$

where, by §61, the order of the partial products is immaterial.]

**63**. By similar processes to those of §53 we see that versors, and therefore also quaternions, are subject to the index-law

$$q^m.q^n = q^{m+n}$$

at least so long as $m$ and $n$ are positive integers.

The extension of this property to negative and fractional exponents must be deferred until we have defined a negative or fractional power of a quaternion.

**64**. We now proceed to the special case of *quadrantal* versors, from whose properties it is easy to deduce all the foregoing results of this chapter. It was, in fact, these properties whose invention by Hamilton in 1843 led almost intuitively to the establishment of the Quaternion Calculus. We will content ourselves at present with an assumption, which will be shown to lead to consistent results ; but at the end of the chapter we will show that no other assumption is possible, following for this purpose a very curious quasi-metaphysical speculation of Hamilton.

**65**. Suppose we have a system of three mutually perpendicular unit-vectors, drawn from one point, which we may call for shortness **i**, **j**, **k**. Suppose also that these are so situated that a positive (i.e. *left-handed*) rotation through a right angle about **i** as an axis brings **j** to coincide with **k**. Then it is obvious that positive quadrantal rotation about **j** will make **k** coincide with **i**; and, about **k**, will make **i** coincide with **j**.

For defniteness we may suppose **i** to be drawn *eastwards*, **j** *northwards*, and **k** *upwards*. Then it is obvious that a positive (left-handed) rotation about the eastward line (**i**) brings the northward line (**j**) into a vertically upward position (**k**) ; and so of the others.

**66**. Now the operator which turns **j** into **k** is a quadrantal versor (§53) ; and, as its axis is the vector **i**, we may call it $i$.

Thus

$$\frac{\mathbf{k}}{\mathbf{j}} = i, \text{ or } \mathbf{k} = i\mathbf{j} \tag{1}$$

Similary we may put

$$\frac{\mathbf{i}}{\mathbf{k}} = j, \text{ or } \mathbf{i} = j\mathbf{k} \tag{2}$$

and

$$\frac{\mathbf{j}}{\mathbf{i}} = k, \text{ or } \mathbf{j} = k\mathbf{i} \tag{3}$$

[It may be here noticed, merely to show the symmetry of the system we arc explaining, that if the three mutually perpendicular vectors **i**, **j**, **k** be made to revolve about a line equally inclined to all, so that **i** is brought to coincide with **j**, **j** will then coincide with **k**, and **k** with **i**: and the above equations will still hold good, only (1) will become (2), (2) will become (3), and (3) will become (1).]

**67**. By the results of §50 we see that

$$\frac{-\mathbf{j}}{\mathbf{k}} = \frac{\mathbf{k}}{\mathbf{j}}$$

i.e. a southward unit- vector bears the same ratio to an upward unit-vector that the latter does to a northward one; and therefore we have

Thus

$$\frac{-\mathbf{j}}{\mathbf{k}} = i, \text{ or } -\mathbf{j} = i\mathbf{k} \tag{4}$$

Similary t

$$\frac{-\mathbf{k}}{\mathbf{i}} = j, \text{ or } -\mathbf{k} = j\mathbf{i} \tag{5}$$

and

$$\frac{-\mathbf{i}}{\mathbf{j}} = k, \text{ or } -\mathbf{i} = k\mathbf{j} \tag{6}$$

**68**. By (4) and (1) we have

$$-j = ik = i(ij) \text{ (by the assumption in §54) } = i^2 j$$

Hence

$$i^2 = -1 \tag{7}$$

Arid in the same way, (5) and (2) give

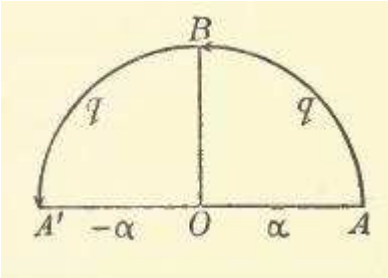
$$j^2 = -1 \tag{8}$$

and (6) and (3)

$$k^2 = -1 \tag{9}$$

Thus, as the directions of **i**, **j**, **k** are perfectly arbitrary, we see that *the square of every quadrantal versor is negative unity*.

[Though the following proof is in principle exactly the same as the foregoing, it may perhaps be of use to the student, in showing him precisely the nature as well as the simplicity of the step we have taken.




Let $ABA'$ be a semicircle, whose centre is 0, and let $OB$ be perpendicular to $AOA'$.

Then $\dfrac{\overline{OB}}{\overline{OA'}} = q$ suppose, is a quadrantal versor, and is evidently equal to $\dfrac{\overline{OA'}}{\overline{OB}}$ ;

§§50, 53. Hence

$$q^2 = \frac{\overline{OA'}}{\overline{OB}} \cdot \frac{\overline{OB}}{\overline{OA}} = \frac{\overline{OA'}}{\overline{OA}} = -1]$$

**69**. Having thus found that the squares of *i*, *j*, *k* are each equal to negative unity ; it only remains that we find the values of their products two and two. For, as we will see, the result is such as to show that the value of any other combination whatever of *i*,*j*, *k* (as factors of a product) may be deduced from the values of these squares and products.

Now it is obvious that

$$\frac{\mathbf{k}}{-\mathbf{i}} = \frac{\mathbf{i}}{\mathbf{k}} = j$$

(i.e. the versor which turns a westward unit-vector into an upward one will turn the upward into an eastward unit) ; or

$$\mathbf{k} = j(-\mathbf{i}) = -j\mathbf{i} \tag{10}$$

Now let us operate on the two equal vectors in (10) by the same versor, $i$, and we have

$$i\mathbf{k} = i(-j\mathbf{i}) = -j\mathbf{i}$$

But by (4) and (3)

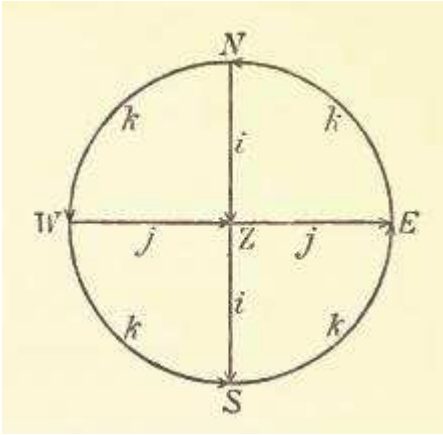
$$i\mathbf{k} = -\mathbf{j} = -k\mathbf{i}$$

Comparing these equations, we have

$$-ij\mathbf{i} = -k\mathbf{i}$$

$$\left.\begin{array}{ll} \text{or, §54 (end),} & ij = k \\ \text{and symmetry gives} & jk = i \\ & ki = j \end{array}\right\} \tag{11}$$

The meaning of these important equations is very simple ; and is, in fact, obvious from our construction in §54 for the multiplication of versors ; as we see by the annexed figure, where we must remember that $i$, $j$, $k$ are quadrantal versors whose planes are at right angles, so that the figure represents a hemisphere divided into quadrantal triangles. [The arrow-heads indicate the direction of each vector arc.]




Thus, to show that $ij = k$, we have, $O$ being the centre of the sphere, $N$, $E$, $S$, $W$ the north, east, south, and west, and $Z$ the zenith (as in §65) ;

$$j\overline{OW} = \overline{OZ}$$

whence $$ij\overline{OW} = i\overline{OZ} = \overline{OS} = k\overline{OW}$$

* The negative sign, being a mere numerical factor, is evidently commutative with $j$ indeed we may, if necessary, easily assure ourselves of the fact that to turn the negative (or reverse) of a vector through a right (or indeed any) angle, is the same thing as to turn the vector through that angle and then reverse it.

**70**. But, by the same figure,

$$i\overline{ON} = \overline{OZ}$$

whence $$ji\overline{ON} = j\overline{OZ} = \overline{OE} = -\overline{OW} = -k\overline{ON}.$$

**71**. From this it appears that

$$\left.\begin{array}{l} ji = -k \\ kj = -i \\ ik = -j \end{array}\right\} \qquad (12)$$

and thus, by comparing (11),

$$\left.\begin{array}{l} ij = -ji = k \\ jk = -kj = i \\ ki = -ik = j \end{array}\right\} \qquad (11), (12)$$

These equations, along with

$$i^2 = j^2 = k^2 = -1 \qquad ((7), (8), (9))$$

contain essentially the whole of Quaternions. But it is easy to see that, for the first group, we may substitute the single equation

$$ijk = -1 \qquad (13)$$

since from it, by the help of the values of the squares of *i*, *j*, *k*, all the other expressions may be deduced. We may consider it proved in this way, or deduce it afresh from the figure above, thus

$$k\overline{ON} = \overline{OW}$$

$$jk\overline{ON} = j\overline{OW} = \overline{OZ}$$

$$ijk\overline{ON} = ij\overline{OW} = i\overline{OZ} = \overline{OS} = -\overline{ON}$$

**72**. One most important step remains to be made, to wit the assumption referred to in §64. We have treated *i*, *j*, *k* simply as quadrantal versors ; and **i**, **j**, **k** as unit-vectors at right angles to each other, and coinciding with the axes of rotation of these versors. But if we collate and compare the equations just proved we have

$$\left\{\begin{array}{ll} i^2 = -1 & (7) \\ \mathbf{i}^2 = -1 & (\S 9) \end{array}\right.$$

$$\left\{\begin{array}{ll} ij = k & (11) \\ i\mathbf{j} = \mathbf{k} & (1) \end{array}\right.$$

$$\left\{\begin{array}{ll} ji = -k & (11) \\ j\mathbf{i} = -\mathbf{k} & (1) \end{array}\right.$$

with the other similar groups symmetrically derived from them.

Now the meanings we have assigned to *i*, *j*, *k* are quite independent of, and not inconsistent with, those assigned to **i**, **j**, **k**. And it is superfluous to use two sets of characters when one will suffice. Hence it appears that *i*, *j*, *k* may be substituted for **i**, **j**, **k**; in other words, *a unit-vector when employed as a factor may be considered as a quadrantal versor whose plane is perpendicular to the vector*. (Of course it follows that every vector can be treated as the product of a number and a quadrantal versor.) This is one of the main elements of the singular simplicity of the quaternion calculus.

**73**. Thus *the product, and therefore the quotient, of two perpendicular vectors is a third vector perpendicular to both.*

Hence the reciprocal (§51) of a vector is a vector which has the *opposite* direction to that of the vector, arid its length is the reciprocal of the length of the vector.

The conjugate (§52) of a vector is simply the vector reversed.

Hence, by §52, if $\alpha$ be a vector

$$(Ta)^2 = \alpha K\alpha = \alpha(-\alpha) = -\alpha^2$$

**74**. We may now see that *every versor may be represented by a power of a unit-vector.*

For, if $\alpha$ be any vector perpendicular to $i$ (which is *any* definite unit-vector), $i\alpha = \beta$ is a vector equal in length to $\alpha$, but perpendicular to both $i$ and $\alpha$

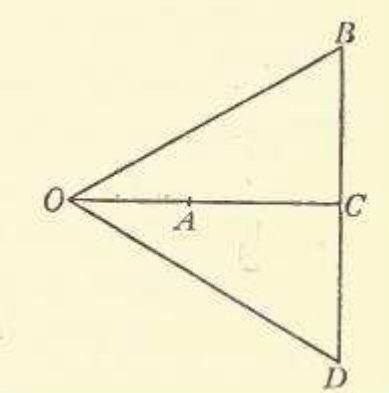
$$\begin{aligned} i^2\alpha &= -\alpha \\ \imath^3\alpha &= -i\alpha = -\beta \\ \imath^4\alpha &= -i\beta = -i^2\alpha = \alpha \end{aligned}$$

Thus, by successive applications of $i$, $\alpha$. is turned round $i$ as an axis through successive right angles. Hence it is natural to *define $i^m$ as a versor which turns any vector perpendicular to i through m right angles in the positive direction of rotation about i as an axis.* Here $m$ may have any real value whatever, whole or fractional, for it is easily seen that analogy leads us to interpret a negative value of $m$ as corresponding to rotation in the negative direction.

**75**. From this again it follows that *any quaternion may be expressed as a power of a vector.* For the tensor and versor elements of the vector may be so chosen that, when raised to the same power, the one may be the tensor and the other the versor of the given quaternion. The vector must be, of course, perpen dicular to the plane of the quaternion.

**76**. And we now see, as an immediate result of the last two sections, that the index-law holds with regard to powers of a quaternion (§63).

**77**. So far as we have yet considered it, a quaternion has been regarded as the *product* of a tensor and a versor: we are now to consider it as a *sum*. The easiest method of so analysing it seems to be the following.




Let $\dfrac{\overline{OB}}{\overline{OA}}$ represent any quaternion. Draw $BC$ perpendicular to $OA$, produced if necessary.

Then, §19, $\overline{OB} = \overline{OC} + \overline{CB}$

But, §22, $\overline{OC} = x\overline{OA}$

where $x$ is a number, whose sign is the same as that of the cosine of $\angle AOB$.

Also, §73, since $CB$ is perpendicular to $OA$,

$$\overline{CB} = \gamma\overline{OA}$$

where $\gamma$ is a vector perpendicular to $OA$ and $CB$, i.e. to the plane of the quaternion; and, as the figure is drawn, directed *towards* the reader.

Hence

$$\frac{\overline{OB}}{\overline{OA}} = \frac{x\overline{OA} + \gamma\overline{OA}}{\overline{OA}} = x + \gamma$$

Thus a quaternion, in general, may be decomposed into the sum of two parts, one numerical, the other a vector. Hamilton calls them the SCALAR, and the VECTOR, and denotes them respectively by the letters $S$ and $V$ prefixed to the expression for the quaternion.

**78**. Hence $q = Sq + Vq$, and if in the above example

$$\frac{\overline{OB}}{\overline{OA}} = q$$

then

$$\overline{OB} = \overline{OC} + \overline{CB} = Sq.\overline{OA} + Vq.\overline{OA}^{2}$$

The equation above gives

$$\overline{OC} = Sq.\overline{OA}$$
$$\overline{CB} = Vq.\overline{OA}$$

**79**. If, in the last figure, we produce $BC$ to $D$, so as to double its length, and join $OD$, we have, by §52,

$$\frac{\overline{OD}}{\overline{OA}} = Kq = SKq + VKq$$

so that $\overline{OD} = \overline{OC} + \overline{CD} = SKq.\overline{OA} + VKq.\overline{OA}$

Hence $\overline{OC} = SKq.\overline{OA}$

and $\overline{CD} = VKq.\overline{OA}$

Comparing this value of $\overline{OC}$ with that in last section, we find

$$SKq = Sq \tag{1}$$

or *the scalar of the conjugate of a quaternion is equal to the scalar of the quaternion.*

Again, $\overline{CD} = -\overline{CB}$ by the figure, and the substitution of their values gives

$$VKq = -Vq \tag{2}$$

or *the vector of the conjugate of a quaternion is the vector of the quaternion reversed.*

We may remark that the results of this section are simple con sequences of the fact that the symbols $S$, $V$, $K$ are commutative [2].

Thus $SKq = KSq = Sq$,

since the conjugate of a number is the number itself; and

$$VKq = KVq = -Vq (\S 73)$$

[2] The points are inserted to show that $S$ and $V$ apply only to $q$, and not to $q\overline{OA}$.

[2] It is curious to compare the properties of these quaternion symbols with those of the Elective Symbols of Logic, as given in BOOLE'S wonderful treatise on the *Laws of Thought*; and to think that the same grand science of mathematical analysis, by processes remarkably similar to each other, reveals to us truths in the science of position far beyond the powers of the geometer, and truths of deductive reasoning to which unaided thought could never have led the logician.

Again, it is obvious that,

$$\sum Sq = S\sum q, \quad \sum Vq = V\sum q$$

and thence $$\sum Kq = K\sum q$$

**80**. Since any vector whatever may be represented by

$$xi + yj + zk$$

where $x$, $y$, $z$ are numbers (or Scalars), and $i$, $j$, $k$ may be any three non-coplanar vectors, §§23, 25 though they are usually understood as representing a rectangular system of unit-vectors and since any scalar may be denoted by $w$; we may write, for any quaternion $q$, the expression

$$q = w + xi + yj + zk(\S 78)$$

Here we have the essential dependence on four distinct numbers, from which the quaternion derives its name, exhibited in the most simple form.

And now we see at once that an equation such as

$$q' = q$$

where $$q' = w' + x'i + y'j + z'k$$
involves, of course, the *four* equations

$$w' = w,\ x' = x,\ y' = y,\ z' = z$$

**81**. We proceed to indicate another mode of proof of the distributive law of multiplication.

We have already defined, or assumed (§61), that

$$\frac{\beta}{\alpha} + \frac{\gamma}{\alpha} = \frac{\beta+\gamma}{\alpha}$$

or $$\beta\alpha^{-1} + \gamma\alpha^{-1} = (\beta+\gamma)\alpha^{-1}$$
and have thus been able to understand what is meant by adding two quaternions.

But, writing $\alpha$ for $\alpha^{-1}$, we see that this involves the equality

$$(\beta+\gamma)\alpha = \beta\alpha + \gamma\alpha$$

from which, by taking the conjugates of both sides, we derive

$$\alpha'(\beta'+\gamma') = \alpha'\beta' + \alpha'\gamma'(\S 55)$$

And a combination of these results (putting $\beta+\gamma$ for $\alpha'$ in the latter, for instance) gives

$$\begin{aligned}(\beta+\gamma)(\beta'+\gamma') &= (\beta+\gamma)\beta' + (\beta+\gamma)\gamma' \\ &= \beta\beta' + \gamma\beta' + \beta\gamma' + \gamma\gamma'\end{aligned}$$

by the former.

Hence the *distributive principle is true in the multiplication of vectors.*

It only remains to show that it is true as to the scalar and vector parts of a quaternion, and then we will easily attain the general proof.

Now, if $a$ be any scalar, $\alpha$ any vector, and $q$ any quaternion,

$$(a+\alpha)q = aq + \alpha q$$

For, if $\beta$ be the vector in which the plane of $q$ is intersected by a plane perpendicular to $\alpha$, we can find other two vectors, $\gamma$ and $\delta$ one in each of these planes such that

$$\alpha = \frac{\gamma}{\beta}, \qquad q = \frac{\beta}{\delta}$$

And, of course, $a$ may be written $\dfrac{a\beta}{\beta}$; so that

$$\begin{aligned}(a+\alpha)q &= \tfrac{a\beta+\gamma}{\beta}.\tfrac{\beta}{\delta} = \tfrac{a\beta+\gamma}{\delta} \\ &= a\tfrac{\beta}{\delta} + \tfrac{\gamma}{\delta} = a\tfrac{\beta}{\delta} + \tfrac{\gamma}{\beta}.\tfrac{\beta}{\delta} \\ &= aq + \alpha q\end{aligned}$$

And the conjugate may be written

$$q'(a'+\alpha') = q'a' + q'\alpha' (\S 55)$$

Hence, generally,

$$(a+\alpha)(b+\beta) = ab + a\beta + b\alpha + \alpha\beta$$

or, breaking up $a$ and $b$ each into the sum of two scalars, and $\alpha$, $\beta$ each into the sum of two vectors,

$(a_1 + a_2 + \alpha_1 + \alpha_2)(b_1 + b_2 + \beta_1 + \beta_2)$

$$= (a_1+a_2)(b_1+b_2) + (a_1+a_2)(\beta_1+\beta_2) + (b_1+b_2)(\alpha_1+\alpha_2) + (\alpha_1+\alpha_2)(\beta_1+\beta_2)$$

(by what precedes, all the factors on the right are distributive, so that we may easily put it in the form)

$$= (a_1+\alpha_1)(b_1+\beta_1) + (a_1+\alpha_1)(b_2+\beta_2) + (a_2+\alpha_2)(b_1+\beta_1) + (a_2+\alpha_2)(b_2+\beta_2)$$

Putting $a_1 + \alpha_1 = p$, $\quad a_2 + \alpha_2 = q$, $\quad b_1 + \beta_1 = r$, $\quad b_2 + \beta_2 = s$,
we have $(p+q)(r+s) = pr + ps + qr + qs$

**82**. Cayley suggests that the laws of quaternion multiplication may be derived more directly from those of vector multiplication, supposed to be already established. Thus, let $\alpha$ be the unit vector perpendicular to the vector parts of $q$ and of $q'$. Then let

$$\rho = q.\alpha, \quad \sigma = -\alpha.q'$$

as is evidently permissible, and we have

$$p\alpha = q.\alpha\alpha = -q; \quad \alpha\sigma = -\alpha\alpha.q' = q'$$

so that

$$-q.q' = \rho\alpha.\alpha\sigma = -\rho.\sigma$$

The student may easily extend this process.

For variety, we will now for a time forsake the geometrical mode of proof we have hitherto adopted, and deduce some of our next steps from the analytical expression for a quaternion given in §80, and the properties of a rectangular system of unit-vectors as in §71.

We will commence by proving the result of §77 anew.

**83**. Let

$$\alpha = xi + yj + zk$$
$$\beta = x'i + y'j + z'k$$

Then, because by §71 every product or quotient of $i$, $j$, $k$ is reducible to one of them or to a number, we are entitled to assume

$$q = \frac{\beta}{\alpha} = \omega + \xi i + \eta j + \zeta k$$

where $\omega$, $\xi$, $\eta$, $\zeta$ are numbers. This is the proposition of §80.

[Of course, with this expression for a quaternion, there is no necessity for a formal proof of such equations as

$$p + (q + r) = (p + q) + r$$

where the various sums are to be interpreted as in §61.

All such things become obvious in view of the properties of $i$, $j$ ,$k$.]

**84**. But it may be interesting to find $\omega$, $\xi$, $\eta$, $\zeta$ in terms of $x$, $y$, $z$, $x'$, $y'$ , $z'$ .

We have

$$\beta = q\alpha$$

or

$$x'i + y'j + z'k = (\omega + \xi i + \eta j + \zeta k)(xi + yj + zk)$$
$$= -(\xi x + \eta y + \zeta z) + (\omega x + \eta z - \zeta y)i + (\omega y + \zeta x - \xi z)j + (\omega z + \xi y - \eta x)k$$

as we easily see by the expressions for the powers and products of $i$, $j$, $k$ given in §71. But the student must pay particular attention to the *order* of the factors, else he is certain to make mistakes.

This (§80) resolves itself into the four equations

$$\begin{array}{lllllllll} 0 & = & & & \xi x & + & \eta y & + & \zeta z \\ x' & = & \omega x & & & + & \eta z & - & \zeta y \\ y' & = & \omega y & - & \xi z & & & + & \zeta x \\ z' & = & \omega z & + & \xi y & - & \eta x & & \end{array}$$

The three last equations give

$$xx' + yy' + zz' = \omega(x^2 + y^2 + z^2)$$

which determines $\omega$.

Also we have, from the same three, by the help of the first,

$$\xi x' + \eta y' + \zeta z' = 0$$

which, combined with the first, gives

$$\frac{\xi}{yz' - zy'} = \frac{\eta}{zx' - xz'} = \frac{\zeta}{xy' - yx'}$$

and the common value of these three fractions is then easily seen to be

$$\frac{1}{x^2+y^2+z^2}$$

It is easy enough to interpret these expressions by means of ordinary coordinate geometry : but a much simpler process will be furnished by quaternions themselves in the next chapter, and, in giving it, we will refer back to this section.

**85**. The associative law of multiplication is now to be proved by means of the distributive (§81). We leave the proof to the student. He has merely to multiply together the factors

$$w+xi+yj+zk, \quad w+x'i+y'j+z'k, \quad \text{and } w''+x''i+y''j+z''k$$

as follows :

First, multiply the third factor by the second, and then multiply the product by the first; next, multiply the second factor by the first and employ the product to multiply the third: always remembering that the multiplier in any product is placed *before* the multiplicand. He will find the scalar parts and the coefficients of $i$, $j$, $k$, in these products, respectively equal, each to each.

**86**. With the same expressions for $\alpha$, $\beta$, as in section 83, we have

$$\alpha\beta = (xi+yj+zk)(x'i+y'j+z'k)$$

$$= -(xx'+yy'+zz') + (yz'-zy')i + (zx'-xz')j + (xy'-yx')k$$

But we have also

$$\beta\alpha = -(xx'+yy'+zz') - (yz'-zy')i - (zx'-xz')j - (xy'-yx')k$$

The only difference is in the sign of the vector parts. Hence

$$S\alpha\beta = S\beta\alpha \tag{1}$$

$$V\alpha\beta = -V\beta\alpha \tag{2}$$

$$\alpha\beta+\beta\alpha = 2S\alpha\beta \tag{3}$$

$$\alpha\beta-\beta\alpha = 2V\alpha\beta \tag{4}$$

$$\alpha\beta = K.\beta\alpha \tag{5}$$

**87**. If $\alpha=\beta$ we have of course (§25)

$$x=x', \quad y=y', \quad z=z'$$

and the formulae of last section become

$$\alpha\beta=\beta\alpha=\alpha^2=-(x^2+y^2+z^2)$$

which was anticipated in §73, where we proved the formula

$$(T\alpha)^2=-\alpha^2$$

and also, to a certain extent, in §25.

**88**. Now let $q$ and $r$ be any quaternions, then

$$\begin{aligned} S.qr &= S.(Sq+Vq)(Sr+Vr) \\ &= S.(SqSr+Sr.Vq+Sq.Vr+VqVr) \\ &= SqSr+S.VqVr \end{aligned}$$

since the two middle terms are vectors. Similarly,

$$S.rq = SrSq+S.VrVq$$

Hence, since by (1) of §86 we have

$$S.VqVr = S.VrVq$$

we see that

$$S.qr = S.rq \tag{1}$$

a formula of considerable importance.

It may easily be extended to any number of quaternions, because, $r$ being arbitrary, we may put for it $rs$. Thus we have

$$\begin{aligned} S.qrs &= S.rsq \\ &= S.sqr \end{aligned}$$

by a second application of the process. In words, we have the theorem *the scalar of the product of any number of given quaternions depends only upon the cyclical order in which they are arranged.*

**89**. An important case is that of three factors, each a vector. The formula then becomes

$$S.\alpha\beta\gamma = S.\beta\gamma\alpha = S.\gamma\alpha\beta$$

But

$$\begin{aligned} S.\alpha\beta\gamma &= S\alpha(S\beta\gamma+V\beta\gamma) & \\ &= S\alpha V\beta\gamma & \text{since } \alpha S\beta\gamma \text{ is a vector} \\ &= -S\alpha V\gamma\beta & \text{by (2) of §86} \\ &= -S\alpha(S\gamma\beta+V\gamma\beta) & \\ &= -S.\alpha\gamma\beta & \end{aligned}$$

Hence *the scalar of the product of three vectors changes sign when the cyclical order is altered.*

By the results of §§55, 73, 79 we see that, for any number of vectors, we have

$$K.\alpha\beta\gamma\ldots\phi\chi = \pm\chi\phi\ldots\gamma\beta\alpha$$

(the positive sign belonging to the product of an even number of vectors) so that

$$S.\alpha\beta\ldots\phi\chi = \pm S.\chi\phi\ldots\beta\alpha$$

Similarly

$$V.\alpha\beta\ldots\phi\chi = \mp V.\chi\phi\ldots\beta\alpha$$

Thus we may generalize (3) and (4) of §86 into

$$2S.\alpha\beta\ldots\phi\chi = \alpha\beta\ldots\chi\phi \pm \phi\chi\ldots\beta\alpha$$

$$2V.\alpha\beta\ldots\phi\chi = \alpha\beta\ldots\chi\phi \mp \phi\chi\ldots\beta\alpha$$

the upper sign still being used when the -number of factors is even.

Other curious propositions connected with this will be given later (some, indeed, will be found in the Examples appended to this chapter), as we wish to develop the really fundamental formulae in as compact a form as possible.

**90**. By (4) of §86,

$$2V\beta\gamma = \beta\gamma - \gamma\beta$$

Hence

$$2V.\alpha V\beta\gamma = V.\alpha(\beta\gamma - \gamma\beta)$$

(by multiplying both by $\alpha$, and taking the vector parts of each side)

$$= V(\alpha\beta\gamma + \beta\alpha\gamma - \beta\alpha\gamma - \alpha\gamma\beta)$$

(by introducing the null term $\beta\alpha\gamma - \beta\alpha\gamma$).

That is

$$2V.\alpha V\beta\gamma = V.(\alpha\beta + \beta\alpha)\gamma - V(\beta S\alpha\gamma + \beta V\alpha\gamma + S\alpha\gamma.\beta + V\alpha\gamma.\beta$$
$$= V.(2S\alpha\beta)\gamma - 2V\beta S\alpha\gamma$$

(if we notice that $V(V\alpha\gamma.\beta) = -V.\beta V\alpha\gamma$ by (2) of §86). Hence

$$V.\alpha V\beta\gamma = \gamma S\alpha\beta - \beta S\gamma\alpha \tag{1}$$

a formula of constant occurrence.

Adding $\alpha S\beta\gamma$ to both sides, we get another most valuable formula

$$V.\alpha\beta\gamma = \alpha S\beta\gamma - \beta S\gamma\alpha + \gamma S\alpha\beta \tag{2}$$

and the form of this shows that we may interchange $\gamma$ and $\alpha$ without altering the right-hand member. This gives

$$V.\alpha\beta\gamma = V.\gamma\beta\alpha$$

a formula which may be greatly extended. (See §89, above.)

Another simple mode of establishing (2) is as follows :

$$\begin{aligned} K.\alpha\beta\gamma &= -\gamma\beta\alpha \\ \therefore 2V.\alpha\beta\gamma &= \alpha\beta\gamma - K.\alpha\beta\gamma \text{ (by §79(2))} \\ &= \alpha\beta\gamma + \gamma\beta\alpha \\ &= \alpha(\beta\gamma + \gamma\beta) - (\alpha\gamma + \gamma\alpha)\beta + \gamma(\alpha\beta + \beta\alpha) \\ &= 2\alpha S\beta\gamma - 2\beta S\alpha\gamma + 2\gamma S\alpha\beta \end{aligned}$$

**91**. We have also

$$VV\alpha\beta V\gamma\delta = -VV\gamma\delta V\alpha\beta \quad \text{by (2) of §86}$$
$$= \delta S\gamma V\alpha\beta - \gamma S\delta V\alpha\beta = \delta S.\alpha\beta\gamma - \gamma S.\alpha\beta\delta$$
$$= -\beta S\alpha V\gamma\delta + \alpha S\beta V\gamma\delta = -\beta S.\alpha\gamma\delta + \alpha S.\beta\gamma\delta$$

all of these being arrived at by the help of §90 (1) and of §89; and by treating alternately $V\alpha\beta$ and $V\gamma\delta$ as *simple* vectors.

Equating two of these values, we have

$$\delta S.\alpha\beta\gamma = \alpha S.\beta\gamma\delta + \beta S.\gamma\alpha\delta + \gamma S.\alpha\beta\delta \tag{3}$$

a very useful formula, expressing any vector whatever in terms of three given vectors. [This, of course, presupposes that $\alpha$, $\beta$, $\gamma$ are not coplanar, §23. In fact, if they be coplanar, the factor $S.\alpha\beta\gamma$ vanishes, and thus (3) does not give an expression for $\delta$. This will be shown in §101 below.]

**92**. That such an expression as (3) is possible we knew already by §23. For variety we may seek another expression of a similar character, by a process which differs entirely from that employed in last section.

$\alpha$, $\beta$, $\gamma$ being any three non-coplanar vectors, we may derive from them three others $V\alpha\beta$, $V\beta\gamma$, $V\gamma\alpha$ and, as these will not be coplanar, any other vector $\delta$ may be expressed as the sum of the three, each multiplied by some scalar. It is required to find this expression for $\delta$.

Let

$$\delta = xV\alpha\beta + yV\beta\gamma + zV\gamma\alpha$$

Then

$$S\gamma\delta = xS.\gamma\alpha\beta = xS.\alpha\beta\gamma$$

the terms in y and z going out, because

$$S\gamma V\beta\gamma = S.\gamma\beta\gamma = S\beta\gamma^2 = \gamma^2 S\beta = 0$$

for $\gamma^2$ is (§73) a number.

Similarly

$$S\beta\delta = zS.\beta\gamma\alpha = zS.\alpha\beta\gamma$$

and

$$S\alpha\delta = qS.\alpha\beta\gamma$$

Thus

$$\delta S.\alpha\beta\gamma = V\alpha\beta S\gamma\delta + V\beta\gamma S\alpha\delta + V\gamma\alpha S\beta\delta \tag{4}$$

**93**. We conclude the chapter by showing (as promised in §64) that the assumption that the product of two parallel vectors is a number, and the product of two perpendicular vectors a third vector perpendicular to both, is not only useful and convenient, but absolutely inevitable, if our system is to deal indifferently with all directions in space. We abridge Hamilton s reasoning.

Suppose that there is no direction in space pre-eminent, and that the product of two vectors is something which has quantity, so as to vary in amount if the factors are changed, and to have its sign changed if that of one of them is reversed ; if the vectors be parallel, their product cannot be, in whole or in part, a vector *inclined* to them, for there is nothing to determine the direction in which it must lie. It cannot be a vector *parallel* to them; for by changing the signs of both factors the product is unchanged, whereas, as the whole system has been reversed, the product vector ought to have been reversed. Hence it must be a number. Again, the product of two perpendicular vectors cannot be wholly or partly a number, because on inverting one of them the sign of that number ought to change; but inverting one of them is simply equivalent to a rotation through two right angles about the other, and (from the symmetry of space) ought to leave the number unchanged. Hence the product of two perpendicular vectors must be a vector, and a simple extension of the same reasoning shows that it must be perpendicular to each of the factors. It is easy to carry this farther, but enough has been said to show the character of the reasoning.

## 4.5 Examples To Chapter 2.

**1**. It is obvious from the properties of polar triangles that any mode of representing versors by the *sides* of a spherical triangle must have an equivalent statement in which they are represented by *angles* in the polar triangle.

Show directly that the product of two versors represented by two angles of a spherical triangle is a third versor represented by the *supplement* of the remaining angle of the triangle ; and determine the rule which connects the *directions* in which these angles are to be measured.

**2**. Hence derive another proof that we have not generally

$$pq = qp$$

**3**. Hence show that the proof of the associative principle, §57, may be made to depend upon the fact that if from any point of the sphere tangent arcs be drawn to a spherical conic, and also arcs to the foci, the inclination of either tangent arc to one of the focal arcs is equal to that of the other tangent arc to the other focal arc.

**4**. Prove the formulae

$$2S.\alpha\beta\gamma = \alpha\beta\gamma - \gamma\beta\alpha$$
$$2V.\alpha\beta\gamma = \alpha\beta\gamma + \gamma\beta\alpha$$

**5**. Show that, whatever odd number of vectors be represented by $\alpha$, $\beta$, $\gamma$ &c., we have always

$$V.\alpha\beta\gamma\delta\epsilon = V.\epsilon\delta\gamma\beta\alpha$$

$$V.\alpha\beta\gamma\delta\epsilon\zeta\eta = V.\eta\zeta\epsilon\delta\gamma\beta\alpha, \text{ \&c.}$$

**6**. Show that

$$S.V\alpha\beta V\beta\gamma V\gamma\alpha = -(S.\alpha\beta\gamma)^2$$
$$V.V\alpha\beta V\beta\gamma V\gamma\alpha = V\alpha\beta(\gamma^2 S\alpha\beta - S\beta\gamma S\gamma\alpha) + \dots$$

and

$$V(V\alpha\beta V.V\beta\gamma V\gamma\alpha) = (\beta S\alpha\gamma - \alpha S\beta\gamma)S.\alpha\beta\gamma$$

**7**. If $\alpha$, $\beta$, $\gamma$ be any vectors at right angles to each other, show that

$$(\alpha^3 + \beta^3 + \gamma^3)S.\alpha\beta\gamma = \alpha^4 V\beta\gamma + \beta^4 V\gamma\alpha + \gamma^4 V\alpha\beta$$
$$(\alpha^{2n-1} + \beta^{2n-1} + \gamma^{2n-1})S.\alpha\beta\gamma = \alpha^{2n} V\beta\gamma + \beta^{2n} V\gamma\alpha + \gamma^{2n} V\alpha\beta$$

**8**. If $\alpha$, $\beta$, $\gamma$ be non-coplanar vectors, find the relations among the six scalars, $x$, $y$, $z$ and $\xi$, $\eta$, $\zeta$ which are implied in the equation

$$x\alpha + y\beta + z\gamma = \xi V\beta\gamma + \eta V\gamma\alpha + \zeta V\alpha\beta$$

**9**. If $\alpha$, $\beta$, $\gamma$ be any three non-coplanar vectors, express any fourth vector, $\delta$, as a linear function of each of the following sets of three derived vectors.

$$V.\gamma\alpha\beta, \quad V.\alpha\beta\gamma, \quad V.\beta\gamma\alpha$$

and

$$V.V\alpha\beta V\beta\gamma V\gamma\alpha, \quad V.V\beta\gamma V\gamma\alpha V\alpha\beta, \quad V.V\gamma\alpha V\alpha\beta V\beta\gamma$$

**10**. Eliminate $\rho$ from the equations

$$S\alpha\rho = a, \quad S\beta\rho = b, \quad S\gamma\rho = c, \quad S\delta\rho = d$$

where $\alpha$, $\beta$, $\gamma$, $\delta$ are vectors, and $a$, $b$, $c$, $d$ scalars.

**11**. In any quadrilateral, plane or gauche, the sum of the squares of the diagonals is double the sum of the squares of the lines joining the middle points of opposite sides.

## 4.6 Interpretations And Transformations

**94**. Among the most useful characteristics of the Calculus of Quaternions, the ease of interpreting its formulae geometrically, and the extraordinary variety of transformations of which the simplest expressions are susceptible, deserve a prominent place. We devote this Chapter to some of the more simple of these, together with a few of somewhat more complex character but of constant occurrence in geometrical and physical investigations. Others will appear in every succeeding Chapter. It is here, perhaps, that the student is likely to feel most strongly the peculiar difficulties of the new Calculus. But on that very account he should endeavour to master them, for the variety of forms which any one formula may assume, though puzzling to the beginner, is of the utmost advantage to the advanced student, not alone as aiding him in the solution of complex questions, but as affording an invaluable mental discipline.

**95**. If we refer again to the figure of §77 we see that

$$OC = OB\cos AOB$$

$$CB = OB\sin AOB$$

Hence if

$$\overline{AB} = \alpha, \quad \overline{OB} = \beta, \quad \text{and } \angle AOB = \theta$$

we have

$$OB = T\beta, \quad OA = T\alpha$$

$$OC = T\beta\cos\theta, \quad CB = T\beta\sin\theta$$

Hence

$$S\frac{\beta}{\alpha} = \frac{OC}{OA} = \frac{T\beta}{T\alpha}\cos\theta$$

Similarly,

$$TV\frac{\beta}{\alpha} = \frac{CB}{OA} = \frac{T\beta}{T\alpha}\sin\theta$$

Hence, if $\eta$ be a unit-vector perpendicular to $\alpha$ and $\beta$, and such that positive rotation about it, through the angle $\theta$, turns $\alpha$ towards $\beta$ or

$$\eta = \frac{U\overline{CB}}{U\overline{OA}} = U\frac{\overline{CB}}{\overline{OA}} = UV\frac{\beta}{\alpha}$$

we have

$$V\frac{\beta}{\alpha} = \frac{T\beta}{T\alpha}\sin\theta.\eta \qquad \text{(See, again, §84)}$$

**96**. In the same way, or by putting

$$\begin{aligned} \alpha\beta &= S\alpha\beta + V\alpha\beta \\ &= S\beta\alpha - V\beta\alpha \\ &= \alpha^2\left(S\tfrac{\beta}{\alpha} - V\tfrac{\beta}{\alpha}\right) \\ &= T\alpha^2\left(-S\tfrac{\beta}{\alpha} + V\tfrac{\beta}{\alpha}\right) \end{aligned}$$

we may show that

$$S\alpha\beta = -T\alpha T\beta\cos\theta$$

$$TV\alpha\beta = T\alpha T\beta\sin\theta$$

and

$$V\alpha\beta = T\alpha T\beta\sin\theta.\eta$$

where

$$\eta = UV\alpha\beta = U(-V\beta\alpha) = UV\frac{\beta}{\alpha}$$

Thus *the scalar of the product of two vectors is the continued product of their tensors and of the cosine of the supplement of the contained angle.*

*The tensor of the vector of the product of two vectors is the con tinued product of their tensors and the sine of the contained angle ; and the versor of the same is a unit-vector perpendicular to both, and such that the rotation about it from the first vector (i. e. the multiplier) to the second is left-handed or positive.*

*Hence also $TV\alpha\beta$ is double the area of the triangle two of whose sides are $\alpha$, $\beta$.*

**97**. (a) In any plane triangle $ABC$ we have

$$\overline{AC} = \overline{AB} + \overline{BC}$$

Hence,

$$\overline{AC}^2 = S.\overline{AC}\,\overline{AC} = S.\overline{AC}(\overline{AB} + \overline{BC})$$

With the usual notation for a plane triangle the interpretation of this formula is

$$b^2 = -bc\cos A - ab\cos C$$

or

$$b = c\cos C + c\cos A$$

(b) Again we have, obviously,

$$\begin{aligned} V.\overline{AB}\,\overline{AC} &= V.\overline{AB}(\overline{AB} + \overline{BC}) \\ &= V.\overline{AB}\,\overline{BC} \end{aligned}$$

or

$$cb\sin A = ca\sin B$$

whence

$$\frac{\sin A}{a} = \frac{\sin B}{b} = \frac{\sin C}{c}$$

These are truths, but not truisms, as we might have been led to fancy from the excessive simplicity of the process employed.

**98**. From §96 it follows that, if $\alpha$ and $\beta$ be both actual (i. e. real and non-evanescent) vectors, the equation

$$S\alpha\beta = 0$$

shows that $\cos\theta = 0$, or that $\alpha$ is *perpendicular* to $\beta$. And, in fact, we know already that the product of two perpendicular vectors is a vector.

Again if

$$V\alpha\beta = 0$$

we must have $\sin\theta = 0$, or $\alpha$ is *parallel* to $\beta$. We know already that the product of two parallel vectors is a scalar.

Hence we see that

$$S\alpha\beta = 0$$

is equivalent to

$$\alpha = V\gamma\beta$$

where $\gamma$ is an undetermined vector; and that

$$V\alpha\beta = 0$$

is equivalent to

$$\alpha = x\beta$$

where $x$ is an undetermined scalar.

**99**. If we write, as in §§83, 84,

$$\alpha = ix + jy + kz$$
$$\beta = ix' + jy' + kz'$$

we have, at once, by §86,

$$\begin{aligned} S\alpha\beta &= -xx' - yy' - zz' \\ &= -rr'\left(\tfrac{x}{r}\tfrac{x'}{r'} + \tfrac{y}{r}\tfrac{y'}{r'} + \tfrac{z}{r}\tfrac{z'}{r'}\right) \end{aligned}$$

where

$$r = \sqrt{x^2 + y^2 + z^2}, \quad r' = \sqrt{x'^2 + y'^2 + z'^2}$$

Also

$$V\alpha\beta = rr'\left\{\frac{yz' - zy'}{rr'}i + \frac{zx' - xz'}{rr'}j + \frac{xy' = yx'}{rr'}k\right\}$$

These express in Cartesian coordinates the propositions we have just proved. In commencing the subject it may perhaps assist the student to see these more familiar forms for the quaternion expressions ; and he will doubtless be induced by their appearance to prosecute the subject, since he cannot fail even at this stage to see how much more simple the quaternion expressions are than those to which he has been accustomed.

**100**. The expression

$$S.\alpha\beta\gamma$$

may be written

$$SV(\alpha\beta)\gamma$$

because the quaternion $\alpha\beta\gamma$ may be broken up into

$$S(\alpha\beta)\gamma + V(\alpha\beta)\gamma$$

of which the first term is a vector.

But, by §96,

$$SV(\alpha\beta)\gamma = T\alpha T\beta \sin\theta S\eta\gamma$$

Here $T\eta = 1$, let $\phi$ be the angle between $\eta$ and $\gamma$, then finally

$$S.\alpha\beta\gamma = -T\alpha T\beta T\gamma \sin\theta\cos\phi$$

But as $\eta$ is perpendicular to $\alpha$ and $\beta$, $T\gamma\cos\phi$ is the length of the perpendicular from the extremity of $\gamma$ upon the plane of $\alpha$, $\beta$. And as the product of the other three factors is (§96) the area of the parallelogram two of whose sides are $\alpha$, $\beta$, we see that the magnitude of $S.\alpha\beta\gamma$, independent of its sign, is *the volume of the parallelepiped of which three coordinate edges are* $\alpha$, $\beta$, $\gamma$; or six times the volume of the pyramid which has $\alpha$, $\beta$, $\gamma$ for edges.

**101**. Hence the equation

$$S.\alpha\beta\gamma = 0$$

if we suppose $\alpha\beta\gamma$ to be actual vectors, shows either that

$$\sin\theta = 0$$

or

$$\cos\phi = 0$$

i. e. *two of the three vectors are parallel*, or *all three are parallel to one plane*.

This is consistent with previous results, for if $\gamma = p\beta$ we have

$$S.\alpha\beta\gamma = pS.\alpha\beta^2 = 0$$

and, if $\gamma$ be coplanar with $\alpha$,$\beta$, we have $\gamma = p\alpha + q\beta$ and

$$S.\alpha\beta\gamma = S.\alpha\beta(p\alpha + q\beta) = 0$$

**102**. This property of the expression $S.\alpha\beta\gamma$ prepares us to find that it is a determinant. And, in fact, if we take $\alpha$,$\beta$ as in §83, and in addition

$$\gamma = ix'' + jy'' + kz''$$

we have at once

$$S.\alpha\beta\gamma = -x''(yz' - zy') - y''(zx' - xz') - z''(xy' - yx')$$

$$= -\begin{vmatrix} x & y & z \\ x' & y' & z' \\ x'' & y'' & z'' \end{vmatrix}$$

The determinant changes sign if we make any two rows change places. This is the proposition we met with before (§89) in the form

$$S.\alpha\beta\gamma = -S.\beta\alpha\gamma = S.\beta\gamma\alpha, \text{ \&c}$$

If we take three new vectors

$$\alpha_1 = ix + jx' + kx''$$
$$\beta_1 = iy + jy' + ky''$$
$$\gamma_1 = iz + jz' + kz''$$

we thus see that they are coplanar if $\alpha$, $\beta$, $\gamma$ are so. That is, if

$$S.\alpha\beta\gamma = 0$$

then

$$S.\alpha_1\beta_1\gamma_1 = 0$$

**103**. We have, by §52,

$$\begin{array}{rcll} (Tq)^2 & = & qKq = (Sq + Vq)(Sq - Vq) & (\S 79) \\ & = & (Sq)^2 - (Vq)^2 & \text{by algebra} \\ & = & (Sq)^2 + (TVq)^2 & (\S 73) \end{array}$$

If $q = \alpha\beta$, we have $Kq = \beta\alpha$, and the formula becomes

$$\alpha\beta.\beta\alpha = \alpha^2\beta^2 = (S\alpha\beta)^2 - (V\alpha\beta)^2$$

In Cartesian coordinates this is

$$(x^2 + y^2 + z^z)(x'^2 + y'^2 + z'^2)$$
$$= (xx' + yy' + zz')^2 + (yz' - zy')^2 + (zx' - xz')^2 + (xy' - yx')^2$$

More generally we have

$$\begin{array}{rcl} (T(qr))^2 & = & (Tq)^2(Tr)^2 \\ & = & (S.qr)^2 - (V.qr)^2 \end{array}$$

If we write

$$q = w + \alpha = w + ix + jy + kz$$
$$r = w' + \beta = w' + ix' + jy' + kz'$$

this becomes

$$(w^2 + x^2 + y^2 + z^2)(w'^2 + x'^2 + y'^2 + z'^2)$$
$$= (ww' - xx' - yy' - zz')^2 + (wx' + w'x + yz' - zy')^2$$
$$= (xy' + w'y + zx' - xz')^2 + (wz' + w'z + xy' - yx')^2$$

a formula of algebra due to Euler.

**104**. We have, of course, by multiplication,

$$(\alpha + \beta)^2 = \alpha^2 + \alpha\beta + \beta\alpha + \beta^2 = \alpha^2 + 2S\alpha\beta + \beta^2 \qquad (\S 86\ (3))$$

Translating into the usual notation of plane trigonometry, this becomes

$$c^2 = a^2 - 2ab\cos C + b^2$$

the common formula.

Again,

$$V.(\alpha+\beta)(\alpha-\beta) = -V\alpha\beta + V\beta\alpha = -2V\alpha\beta \qquad (\S86\ (2)$$

Taking tensors of both sides we have the theorem, *the parallelogram whose sides are parallel and equal to the diagonals of a given parallelogram, has double its area* (§96).

Also

$$S(\alpha+\beta)(\alpha-\beta) = \alpha^2 - \beta^2$$

and vanishes only when $\alpha^2 = \beta^2$, or $T\alpha = T\beta$; that is, *the diagonals of a parallelogram are at right angles to one another, when, and only when, it is a rhombus.*

Later it will be shown that this contains a proof that the angle in a semicircle is a right angle.

**105**. The expression $\rho = \alpha\beta\alpha^{-1}$
obviously denotes a vector whose tensor is equal to that of $\beta$.

But we have $S.\beta\alpha\rho = 0$
so that $\rho$ is in the plane of $\alpha$, $\beta$

Also we have $S\alpha\rho = S\alpha\beta$
so that $\beta$ and $\rho$ make equal angles with $\alpha$, evidently on opposite sides of it. Thus if $\alpha$ be the perpendicular to a reflecting surface and $\beta$ the path of an incident ray, $-\rho$ will be the path of the reflected ray.

Another mode of obtaining these results is to expand the above expression, thus, §90 (2),

$$\begin{aligned} \rho &= 2\alpha^{-1}S\alpha\beta - \beta \\ &= 2\alpha^{-1}S\alpha\beta - \alpha^{-1}(S\alpha\beta + V\alpha\beta) \\ &= \alpha^{-1}(S\alpha\beta - V\alpha\beta) \end{aligned}$$

so that in the figure of §77 we see that if $\overline{OA} = \alpha$, and $\overline{OB} = \beta$, we have $\overline{OD} = \rho = \alpha\beta\alpha^{-1}$
Or, again, we may get the result at once by transforming the equation to $\frac{\rho}{\alpha} = K(\alpha^{-1}\rho) = K\frac{\beta}{\alpha}$

**106**. For any three coplanar vectors the expression

$$\rho = \alpha\beta\gamma$$

is (§101) a vector. It is interesting to determine what this vector is. The reader will easily see that if a circle be described about the triangle, two of whose sides are (in order) $\alpha$ and $\beta$, and if from the extremity of $\beta$ a line parallel to $\gamma$ be drawn, again cutting the circle, the vector joining the point of intersection with the origin of $\alpha$ is the direction of the vector $\alpha\beta\gamma$. For we may write it in the form

$$\rho = \alpha\beta^2\beta^{-1}\gamma = -(T\beta)^2\alpha\beta^{-1}\gamma = -(T\beta)^2\frac{\alpha}{\beta}\gamma$$

which shows that the versor $\left(\frac{\alpha}{\beta}\right)$ which turns $\beta$ into a direction parallel to $\alpha$, turns $\gamma$ into a direction parallel to $\rho$. And this expresses the long-known property of opposite angles of a quadrilateral inscribed in a circle.

Hence if $\alpha$, $\beta$, $\gamma$ be the sides of a triangle taken in order, the tangents to the circumscribing circle at the angles of the triangle are parallel respectively to

$$\alpha\beta\gamma, \quad \beta\gamma\alpha, \quad \text{and } \gamma\alpha\beta$$

Suppose two of these to be parallel, i. e. let

$$\alpha\beta\gamma = x\beta\gamma\alpha = x\alpha\gamma\beta \quad (\S 90)$$

since the expression is a vector. Hence

$$\beta\gamma = x\gamma\beta$$

which requires either

$$x = 1, \quad V\gamma\beta = 0 \quad \text{or } \gamma||\beta$$

a case not contemplated in the problem; or

$$x = -1, \quad S\beta\gamma = 0$$

i. e. the triangle is right-angled. And geometry shows us at once that this is correct.

Again, if the triangle be isosceles, the tangent at the vertex is parallel to the base. Here we have

$$x\beta = \alpha\beta\gamma$$

or

$$x(\alpha + \gamma) = \alpha(\alpha + \gamma)\gamma$$

whence $x = \gamma^2 = \alpha^2$, or $T\gamma = T\alpha$, as required.

As an elegant extension of this proposition the reader may prove that the vector of the continued product $\alpha\beta\gamma\delta$ of the vectorsides of any quadrilateral inscribed in a sphere is parallel to the radius drawn to the corner ($\alpha$, $\delta$). [For, if $\epsilon$ be the vector from $\delta$, $\alpha$ to $\beta$, $\gamma$, $\alpha\beta\epsilon$ and $\epsilon\gamma\delta$ are (by what precedes) vectors *touching* the sphere at $\alpha$, $\delta$. And their product (whose vector part must be parallel to the radius at $\alpha$, $\delta$) is

$$\alpha\beta\epsilon.\epsilon\gamma\delta = \epsilon^2.\alpha\beta\gamma\delta]$$

**107**. To exemplify the variety of possible transformations even of simple expressions, we will take cases which are of frequent occurrence in applications to geometry.

Thus

$$T(\rho + \alpha) = T(\rho - \alpha)$$

[which expresses that if

$$\overline{OA} = \alpha \quad \overline{OA'} = -\alpha \quad \text{and} \quad \overline{OP} = \rho$$

we have $AP = A'P$

and thus that $P$ is any point equidistant from two fixed points,] may be written

$$(\rho + \alpha)^2 = (\rho - \alpha)^2$$

or $$\rho^2 + 2S\alpha\rho + \alpha^2 = \rho^2 - 2S\alpha\rho + \alpha^2 \quad (\S 104)$$

whence $$S\alpha\rho = 0$$

This may be changed to

$$\alpha\rho + \rho\alpha = 0$$

or

$$\alpha\rho + K\alpha\rho = 0$$

$$SU\frac{\rho}{\alpha} = 0$$

or finally,

$$TVU\frac{\rho}{\alpha} = 1$$

all of which express properties of a plane.

Again, $T\rho = T\alpha$

may be written $T\frac{\rho}{\alpha} = 1$

$$\left(S\frac{\rho}{\alpha}\right)^2 - \left(V\frac{\rho}{\alpha}\right)^2 = 1$$

$$(\rho+\alpha)^2 - 2S\alpha(\rho+\alpha) = 0$$

$$\rho = (\rho+\alpha)^{-1}\alpha(\rho+\alpha)$$

$$S(\rho+\alpha)(\rho-\alpha) = 0$$

or finally,

$$T.(\rho+\alpha)(\rho-\alpha) = 2TV\alpha\rho$$

All of these express properties of a sphere. They will be interpreted when we come to geometrical applications.

**108**. *To find the space relation among five points.*

A system of five points, so far as its internal relations are concerned, is fully given by the vectors from one to the other four. If three of these be called $\alpha$, $\beta$, $\gamma$, the fourth, $\delta$, is necessarily expressible as $x\alpha + y\beta + z\gamma$. Hence the relation required must be independent of x, y, z.

But

$$\left.\begin{array}{rcllll} S\alpha\delta & = & x\alpha^2 & + yS\alpha\beta & + zS\alpha\gamma \\ S\beta\delta & = & xS\beta\alpha & + y\beta^2 & + zS\beta\gamma \\ S\gamma\delta & = & xS\gamma\alpha & + yS\gamma\beta & + z\gamma^2 \\ S\delta\delta = \delta^2 & = & xS\delta\alpha & + yS\delta\beta & + zS\delta\gamma \end{array}\right\} \qquad (1)$$

The elimination of $x$, $y$, $z$ gives a determinant of the fourth order, which may be written

$$\begin{vmatrix} S\alpha\alpha & S\alpha\beta & S\alpha\gamma & S\alpha\delta \\ S\beta\alpha & S\beta\beta & S\beta\gamma & S\beta\delta \\ S\gamma\alpha & S\gamma\beta & S\gamma\gamma & S\gamma\delta \\ S\delta\alpha & S\delta\beta & S\delta\gamma & S\delta\delta \end{vmatrix} = 0$$

Now each term may be put in either of two forms, thus

$$S\beta\gamma = \frac{1}{2}\left\{\beta^2 + \gamma^2 - (\beta-\gamma)^2\right\} = -T\beta T\gamma\cos\widehat{\beta\gamma}$$

If the former be taken we have the expression connecting the distances, two and two, of five points in the form given by Muir (Proc. R. S. E. 1889) ; if we use the latter, the tensors divide out (some in rows, some in columns), and we have the relation among the cosines of the sides and diagonals of a spherical quadrilateral.

We may easily show (as an exercise in quaternion manipulation merely) that this is the *only* condition, by showing that from it we can get the condition when any other of the points is

taken as origin. Thus, let the origin be at $\alpha$, the vectors are $\alpha$, $\beta-\alpha$, $\gamma-\alpha$, $\delta-\alpha$. But, by changing the signs of the first row, and first column, of the determinant above, and then adding their values term by term to the other rows and columns, it becomes

$$\begin{vmatrix} S(\ -\alpha)(-\alpha) & S(\ -\alpha)(\beta-\alpha) & S(\ -\alpha)(\gamma-\alpha) & S(\ -\alpha)(\delta-\alpha) \\ S(\beta-\alpha)(-\alpha) & S(\beta-\alpha)(\beta-\alpha) & S(\beta-\alpha)(\gamma-\alpha) & S(\beta-\alpha)(\delta-\alpha) \\ S(\gamma-\alpha)(-\alpha) & S(\gamma-\alpha)(\beta-\alpha) & S(\gamma-\alpha)(\gamma-\alpha) & S(\gamma-\alpha)(\delta-\alpha) \\ S(\delta-\alpha)(-\alpha) & S(\delta-\alpha)(\beta-\alpha) & S(\delta-\alpha)(\gamma-\alpha) & S(\delta-\alpha)(\delta-\alpha) \end{vmatrix}$$

which, when equated to zero, gives the same relation as before. [See Ex. 10 at the end of this Chapter.]

An additional point, with $\epsilon = x'\alpha + y'\beta + z'\gamma$ gives six additional equations like (1) ; i. e.

$$\begin{array}{rlll} S\alpha\epsilon & = x'\alpha^2 & +y'S\alpha\beta & +z'S\alpha\gamma \\ S\beta\epsilon & = x'S\beta\alpha & +y'\beta^2 & +z'S\beta\gamma \\ S\gamma\epsilon & = x'S\gamma\alpha & +y'S\gamma\beta & +z'\gamma^2 \\ S\delta\epsilon & = x'S\delta\alpha & +y'S\delta\beta & +z'S\delta\gamma \\ & = xS\epsilon\alpha & +yS\epsilon\beta & +zS\epsilon\gamma \\ \epsilon^2 & = x'S\alpha\epsilon & +y'S\beta\epsilon & +z'S\gamma\epsilon \end{array}$$

from which corresponding conclusions may be drawn.

Another mode of solving the problem at the head of this section is to write the *identity*

$$\sum m(\alpha-\theta)^2 = \sum m\alpha^2 - sS.\theta\sum m\alpha + \theta^2\sum m$$

where the $m$s are undetermined scalars, and the $\alpha$s are given vectors, while $\theta$ is any vector whatever.

Now, *provided that the number of given vectors exceeds four*, we do not completely determine the ms by imposing the conditions

$$\sum m = 0, \quad \sum m\alpha = 0$$

Thus we may write the above identity, for each of five vectors successively, as

$$\begin{array}{rcl} \sum m(\alpha-\alpha_1)^2 & = & \sum m\alpha^2 \\ \sum m(\alpha-\alpha_2)^2 & = & \sum m\alpha^2 \\ \ldots\ldots & = & \ldots \\ \sum m(\alpha-\alpha_n)^2 & = & \sum m\alpha^2 \end{array}$$

Take, with these, $\sum m = 0$
and we have six linear equations from which to eliminate the $m$s. The resulting determinant is

$$\begin{vmatrix} \overline{\alpha_1-\alpha_1^2} & \overline{\alpha_1-\alpha_s^2} & \overline{\alpha_1-\alpha_3^2} & . & \overline{\alpha_1-\alpha_5^2} & 1 \\ \overline{\alpha_2-\alpha_1^2} & \overline{\alpha_2-\alpha_s^2} & \overline{\alpha_2-\alpha_3^2} & . & \overline{\alpha_2-\alpha_5^2} & 1 \\ . & . & . & & . & \\ . & . & . & & . & \\ \overline{\alpha_5-\alpha_1^2} & \overline{\alpha_5-\alpha_s^2} & \overline{\alpha_5-\alpha_3^2} & . & \overline{\alpha_5-\alpha_5^2} & 1 \\ 1 & 1 & . & . & 1 & 0 \end{vmatrix} \sum m\alpha^2 = 0$$

This is equivalent to the form in which Cayley gave the relation among the mutual distances of five points. (Camb. Math. Journ. 1841.)

**109**. We have seen in §95 that a quaternion may be divided into its scalar and vector parts as follows:

$$\frac{\beta}{\alpha} = S\frac{\beta}{\alpha} + V\frac{\beta}{\alpha} = \frac{T\beta}{T\alpha}(\cos\theta + \epsilon\sin\theta)$$

where $\theta$ is the angle between the directions of $\alpha$ and $\beta$ and $\epsilon = UV\dfrac{\beta}{\alpha}$ is the unit-vector perpendicular to the plane of $\alpha$ and $\beta$ so situated that positive (i.e. left-handed) rotation about it turns $\alpha$ towards $\beta$

Similarly we have (§96)

$$\begin{aligned}\alpha\beta &= S\alpha\beta + V\alpha\beta \\ &= T\alpha T\beta(-\cos\theta + \epsilon\sin\theta)\end{aligned}$$

$\theta$ and $\epsilon$ having the same signification as before.

**110**. Hence, considering the versor parts alone, we have

$$U\frac{\beta}{\alpha} = \cos\theta + \epsilon\sin\theta$$

Similarly

$$U\frac{\gamma}{\beta} = \cos\phi + \epsilon\sin\phi$$

$\phi$ being the positive angle between the directions of $\gamma$ and $\beta$, and $\epsilon$ the same vector as before, if $\alpha$, $\beta$, $\gamma$ be coplanar.

Also we have

$$U\frac{\gamma}{\alpha} = \cos(\theta+\phi) + \epsilon\sin(\theta+\phi)$$

But we have always

$$\frac{\gamma}{\beta}.\frac{\beta}{\alpha} = \frac{\gamma}{\alpha}$$

and therefore

$$U\frac{\gamma}{\beta}.U\frac{\beta}{\alpha} = U\frac{\gamma}{\alpha}$$

or

$$\begin{aligned}\cos(\phi+\theta) + \epsilon\sin(\phi+\theta) &= (\cos\phi + \epsilon\sin\phi)(\cos\theta + \epsilon\sin\theta) \\ &= \cos\phi\cos\theta - \sin\phi\sin\theta + \epsilon(\sin\phi\cos\theta + \cos\phi\sin\theta)\end{aligned}$$

from which we have at once the fundamental formulae for the cosine and sine of the sum of two arcs, by equating separately the scalar and vector parts of these quaternions.

And we see, as an immediate consequence of the expressions above, that

$$\cos m\theta + \epsilon\sin m\theta = (\cos\theta + \epsilon\sin\theta)^m$$

if $m$ be a positive whole number. For the left-hand side is a versor which turns through the angle $m\theta$ at once, while the right-hand side is a versor which effects the same object by $m$ successive turn ings each through an angle $\theta$. See §§8, 9.

**111**. To extend this proposition to fractional indices we have only to write $\dfrac{\theta}{n}$ for $\theta$, when we obtain the results as in ordinary trigonometry.

From De Moivre's Theorem, thus proved, we may of course deduce the rest of Analytical Trigonometry. And as we have already deduced, as interpretations of self-evident quaternion

transformations (§§97, 104), the fundamental formulae for the solution of plane triangles, we will now pass to the consideration of spherical trigonometry, a subject specially adapted for treatment by qua ternions; but to which we cannot afford more than a very few sections. (More on this subject will be found in Chap. XI in connexion with the Kinematics of rotation.) The reader is referred to Hamilton s works for the treatment of this subject by quaternion exponentials.

**112**. Let $\alpha$, $\beta$, $\gamma$ be unit-vectors drawn from the centr to the corners $A$, $B$, $C$ of a triangle on the unit-sphere. Then it is evident that, with the usual notation, we have (§96),

$$S\alpha\beta = -\cos c, \quad S\beta\gamma = -\cos a, \quad S\gamma\alpha = -\cos b$$

$$TV\alpha\beta = \sin c, \quad TV\beta\gamma = \sin a, \quad TV\gamma\alpha = \sin b$$

Also $UV\alpha\beta$, $UV\beta\gamma$, $UV\gamma\alpha$ are evidently the vectors of the corners of the polar triangle.

Hence

$$S.UV\alpha\beta UV\beta\gamma = \cos B, \text{ \&c.}$$

$$TV.UV\alpha\beta UV\beta\gamma = \sin B, \text{ \&c.}$$

Now (§90 (1)) we have

$$\begin{aligned} SV\alpha\beta V\beta\gamma &= S.\alpha V(\beta V\beta\gamma) \\ &= -S\alpha\beta S\beta\gamma + \beta^2 S\alpha\gamma \end{aligned}$$

Remembering that we have

$$SV\alpha\beta V\beta\gamma = TV\alpha\beta TV\beta\gamma S.UV\alpha\beta UV\beta\gamma$$

we see that the formula just written is equivalent to

$$\sin a \sin c \cos B = -\cos a \cos c + \cos b$$

or

$$\cos b = \cos a \cos c + \sin a \sin c \cos B$$

**113**. Again,

$$V.V\alpha\beta V\beta\gamma = -\beta S\alpha\beta\gamma$$

which gives

$$TV.V\alpha\beta V\beta\gamma = TS.\alpha\beta\gamma = TS.\alpha V\beta\gamma = TS.\beta V\gamma\alpha = TS.\gamma V\alpha\beta$$

or

$$\sin a \sin c \sin B = \sin a \sin p_a = \sin b \sin p_b = \sin c \sin p_c$$

where $p_a$ is the arc drawn from $A$ perpendicular to $BC$, &c. Hence

$$\sin p_a = \sin c \sin B$$

$$\sin p_b = \frac{\sin a \sin c}{\sin b} \sin B$$

$$\sin p_c = \sin a \sin B$$

**114**. Combining the results of the last two sections, we have

$$V\alpha\beta.V\beta\gamma = \sin a \sin c \cos B - \beta \sin a \sin c \sin B$$

$$= \sin a \sin c(\cos B - \beta \sin B)$$

$$\left.\begin{array}{ll} \text{Hence} & U.V\alpha\beta V\beta\gamma = (\cos B - \beta \sin B) \\ \text{and} & U.V\gamma\beta V\beta\alpha = (\cos B + \beta \sin B) \end{array}\right\}$$

These are therefore versors which turn all vectors perpendicular to $OB$ negatively or positively about $OB$ through the angle $B$.

[It will be shown later (§119) that, in the combination

$$(\cos B + \beta \sin B)(\quad)(\cos B - \beta \sin B)$$

the system operated on is made to rotate, as if rigid, round the vector axis $\beta$ through an angle $2B$.]

As another instance, we have

$$\begin{aligned} \tan B &= \frac{\sin B}{\cos B} \\ &= \frac{TV.V\alpha\beta V\beta\gamma}{S.V\alpha\beta V\beta\gamma} \\ &= -\beta^{-1}\frac{V.V\alpha\beta V\beta\gamma}{S.V\alpha\beta V\beta\gamma} \\ &= -\frac{S.\alpha\beta\gamma}{S\alpha\gamma + S\alpha\beta S\beta\gamma} = \text{ \&c} \end{aligned} \tag{1}$$

The interpretation of each of these forms gives a different theorem in spherical trigonometry.

**115**. Again, let us square the equal quantities

$$V.\alpha\beta\gamma \quad \text{and} \quad \alpha S\beta\gamma - \beta S\alpha\gamma + \gamma S\alpha\beta$$

supposing $\alpha$, $\beta$, $\gamma$ to be any unit-vectors whatever. We have

$$-(V.\alpha\beta\gamma)^2 = S^2\beta\gamma + S^2\gamma\alpha + S^2\alpha\beta + 2S\beta\gamma S\gamma\alpha S\alpha\beta$$

But the left-hand member may be written as

$$T^2.\alpha\beta\gamma - S^2.\alpha\beta\gamma$$

whence

$$1 - S^2.\alpha\beta\gamma = S^2\beta\gamma + S^2\gamma\alpha + S^2\alpha\beta + 2S\beta\gamma S\gamma\alpha S\alpha\beta$$

or

$$\begin{aligned} 1 - \cos^2 a - \cos^2 b - \cos^2 c + 2\cos a \cos b \cos c \\ = \sin^2 a \sin^2 p_a = \text{ \&c.} \\ = \sin^2 a \sin^2 b \sin^2 C = \text{ \&c.} \end{aligned}$$

all of which are well-known formulae.

**116**. Again, for any quaternion,

$$q = Sq + Vq$$

so that, if $n$ be a positive integer,

$$q^n = (Sq)^n + n(Sq)^{n-1}Vq + \frac{n.\overline{n-1}}{1.2}(Sq)^{n-2}(Vq)^2 + \dots$$

From this at once

$$S.q^n = (Sq)^n - \frac{n.\overline{n-1}}{1.2}(Sq)^{n-2}T^2Vq$$
$$+\frac{n.\overline{n-1}.\overline{n-2}.\overline{n-3}}{1.2.3.4}(Sq)^{n-4}T^4(Vq) - \&\text{c.},$$
$$V.q^n = Vq\left[n(Sq)^{n-1} - \frac{n.\overline{n-1}.\overline{n-2}}{1.2.3}(Sq)^{n-3}T^2Vq + \&\text{c.},\right]$$

If $q$ be a versor we have

$$q = \cos u + \theta \sin u$$

so that

$$\begin{aligned} S.q^n &= (\cos u)^n - \frac{n.\overline{n-1}}{1.2}(\cos u)^{n-2}(\sin u)^2 + \dots \\ &= \cos nu; \\ V.q^n &= \theta \sin u\left[n(\cos u)^{n-1} - \frac{n.\overline{n-1}.\overline{n-2}}{1.2.3}(\cos u)^{n-3}(\sin u)^2 + \dots\right] \\ &= \theta \sin nu; \end{aligned}$$

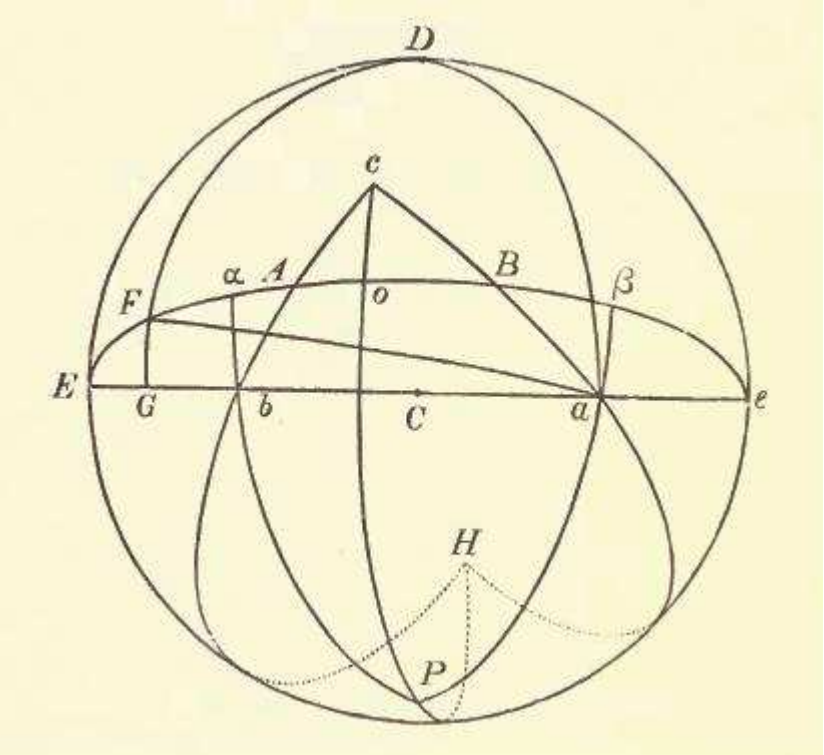
as we might at once have concluded from §110.

Such results may be multiplied indefinitely by any one who has mastered the elements of quaternions.

**117**. A curious proposition, due to Hamilton, gives us a quaternion expression for the *spherical excess* in any triangle. The following proof, which is very nearly the same as one of his, though by no means the simplest that can be given, is chosen here because it incidentally gives a good deal of other information. We leave the quaternion proof as an exercise.

Let the unit-vectors drawn from the centre of the sphere to $A$, $B$, $C$, respectively, be $\alpha$, $\beta$, $\gamma$. It is required to express, as an arc and as an angle on the sphere, the quaternion

$$\beta\alpha^{-1}\gamma$$




The figure represents an orthographic projection made on a plane perpendicular to $\gamma$. Hence $C$ is the centre of the circle $DEe$. Let the great circle through $A$, $B$ meet $DEe$ in $E$, $e$, and let $DE$ be a quadrant. Thus $\overset{\frown}{DE}$ represents $\gamma$ (§72). Also make $\overset{\frown}{EF} = \overset{\frown}{AB} = \beta\alpha^{-1}$ Then, evidently,

$$\overset{\frown}{DF} = \beta\alpha^{-1}\gamma$$

which gives the arcual representation required.

Let $DF$ cut $Ee$ in $G$. Make $Ca = EG$, and join $D$, $a$, and $a$, $F$. Obviously, as $D$ is the pole of $Ee$, $Da$ is a quadrant ; and since $EG = Ca$, $Ga = EG$, a quadrant also. Hence $a$ is the pole of $DG$, and therefore the quaternion may be represented by the angle $DaF$.

Make $Cb = Ca$, and draw the arcs $Pa\beta$, $Pb\alpha$ from $P$, the pole of $AB$. Comparing the triangles $Eb\alpha$ and $ea\beta$, we see that $E\alpha = e\beta$. But, since $P$ is the pole of $AB$, $F\beta a$ is a right angle: and therefore as $Fa$ is a quadrant, so is $F\beta$. Thus $AB$ is the complement of $E\alpha$ or $\beta e$, and therefore

$$\alpha\beta = 2AB$$

Join $bA$. and produce it to $c$ so that $Ac = bA$; join $c$, $P$, cutting $AB$ in $o$. Also join $c$, $B$, and $B$, $a$.

Since $P$ is the pole of $AB$, the angles at $o$ are right angles; and therefore, by the equal triangles $b\alpha A$, $coA$, we have

$$\alpha A = Ao$$

But

$$\alpha\beta = 2AB$$

whence

$$oB = B\beta$$

and therefore the triangles $coB$ and $Ba\beta$ are equal, and $c$, $B$, $a$ lie on the same great circle.

Produce $cA$ and $cB$ to meet in $H$ (on the opposite side of the sphere). $H$ and $c$ are diametrically opposite, and therefore $cP$, produced, passes through $H$.

Now $Pa = Pb = PH$, for they differ from quadrants by the equal arcs $a\beta$, $b\alpha$, $oc$. Hence these arcs divide the triangle $Hab$ into three isosceles triangles.

But

$$\angle PHb + \angle PHA = \angle aHb = \angle bca$$

Also

$$\angle Pab = \pi - \angle cab - \angle PaH$$

$$\angle Pba = \angle Pab = \pi - \angle cba - \angle PbH$$

Adding,

$$2\angle Pab = 2\pi - \angle cab - \angle cba - \angle bca$$

$$= \pi - (\text{spherical excess of } abc)$$

But, as $\angle Fa\beta$ and $\angle Dae$ are right angles, we have

$$\text{angle of } \beta\alpha^{-1}\gamma = \angle FaD = \beta ae = \angle Pab$$

$$= \frac{\pi}{2} - \frac{1}{2}(\text{spherical excess of } abc)$$

[Numerous singular geometrical theorems, easily proved *ab initio* by quaternions, follow from this: e.g. The arc $AB$, which bisects two sides of a spherical triangle $abc$, intersects the base at the distance of a quadrant from its middle point. All spherical triangles, with a common side, and having their other sides bisected by the same great circle (i.e. having their vertices in a small circle parallel to this great circle) have equal areas, &c. ]

**118**. Let $\overline{Oa} = \alpha'$, $\overline{Ob} = \beta'$, $\overline{Oc} = \gamma'$, and we have

$$\begin{aligned}\left(\frac{\alpha'}{\beta'}\right)^{\frac{1}{2}}\left(\frac{\beta'}{\gamma'}\right)^{\frac{1}{2}}\left(\frac{\gamma'}{\alpha'}\right)^{\frac{1}{2}} &= \widehat{Ca}.\widehat{cA}.\widehat{Bc} \\ &= \widehat{Ca}.\widehat{BA} \\ &= \widehat{EG}.\widehat{FE} = \widehat{FG}\end{aligned}$$

But $FG$ is the complement of $DF$. Hence the *angle of the quaternion*

$$\left(\frac{\alpha'}{\beta'}\right)^{\frac{1}{2}}\left(\frac{\beta'}{\gamma'}\right)^{\frac{1}{2}}\left(\frac{\gamma'}{\alpha'}\right)^{\frac{1}{2}}$$

*is half the spherical excess of the triangle whose angular points are at the extremities of the unit-vectors* $\alpha'$, $\beta'$, and $\gamma'$.

[In seeking a purely quaternion proof of the preceding proposi tions, the student may commence by showing that for any three unit-vectors we have

$$\frac{\beta}{\alpha}\frac{\gamma}{\beta}\frac{\alpha}{\gamma} = -(\beta\alpha^{-1}\gamma)^2$$

The angle of the first of these quaternions can be easily assigned; and the equation shows how to find that of $\beta\alpha^{-1}\gamma$.

Another easy method is to commence afresh by forming from the vectors of the corners of a spherical triangle three new vectors thus:

$$\alpha' = \left(\frac{\beta+\gamma}{\alpha}^2\right)^2 . \alpha, \qquad \&c.$$

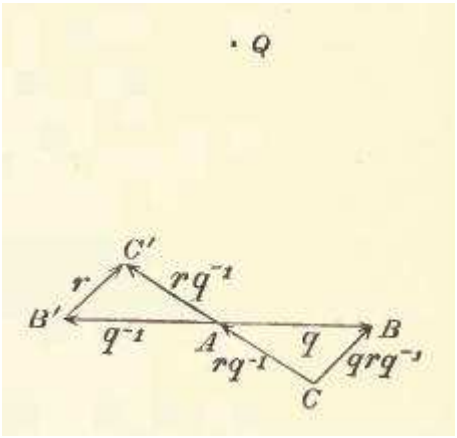
Then the angle between the planes of $\alpha$, $\beta'$ and $\gamma'$, $\alpha$; or of $\beta$, $\gamma'$ and $\alpha'$, $\beta$; or of $\gamma$, $\alpha'$ and $\beta'$, $\gamma$ is obviously the spherical excess.

But a still simpler method of proof is easily derived from the composition of rotations.]

**119**. It may be well to introduce here, though it belongs rather to Kinematics than to Geometry, the interpretation of the operator

$$q(\quad)q^{-1}$$

By a rotation, about the axis of $q$, through double the angle of $q$, the quaternion $r$ becomes the quaternion $qrq^{-1}$ . Its tensor and angle remain unchanged, its plane or axis alone varies.




A glance at the figure is sufficient for the proof, if we note that of course $T.qrq^{-1} = Tr$, and therefore that we need consider the *versor* parts only. Let $Q$ be the pole of $q$.

$$\overset{\frown}{AB} = q, \quad \overset{\frown}{AB}^{-1} = q^{-1}, \quad \overset{\frown}{B'C'} = r$$

Join $C'A$, and make $\overset{\frown}{AC} = \overset{\frown}{C'A}$. Join $CB$.

Then $\overset{\frown}{CB}$ is $qrq^{-1}$, its arc $CB$ is evidently equal in length to that of $r$, $B'C'$; and its plane (making the same angle with $B'B$ that that of $B'C'$ does) has evidently been made to revolve about $Q$, the pole of $q$, through double the angle of $q$.

It is obvious, from the nature of the above proof, that this operation is distributive; i.e. that

$$q(r+s)q^{-1} = qrq^{-1} + qsq^{-1}$$

If $r$ be a vector, $= \rho$, then $q\rho q^{-1}$ (which is also a vector) is the result of a rotation through double the angle of $q$ about the axis of $q$. Hence, as Hamilton has expressed it, if $B$ represent

a rigid system, or assemblage of vectors,

$$qBq^{-1}$$

is its new position after rotating through double the angle of $q$ about the axis of $q$.

**120**. To compound such rotations, we have

$$r.qBq^{-1}.r^{-1} = rq.B.(rq)^{-1}$$

To cause rotation through an angle $t$-fold the double of the angle of $q$ we write

$$q^t Bq^{-t}$$

To reverse the direction of this rotation write

$$q^{-t} Bq^t$$

To *translate* the body $B$ without rotation, each point of it moving through the vector $\alpha$, we write $\alpha + B$.

To produce rotation of the translated body about the same axis, and through the same angle, as before,

$$q(\alpha + B)q^{-1}$$

Had we rotated first, and then translated, we should have had

$$\alpha + qBq^{-1}$$

From the point of view of those who do not believe in the Moon s rotation, the former of these expressions ought to be

$$q\alpha q^{-1} + B$$

instead of

$$q\alpha q^{-1} + qBq^{-1}$$

But to such men quaternions are unintelligible.

**121**. The operator above explained finds, of course, some of its most direct applications in the ordinary questions of Astronomy, connected with the apparent diurnal rotation of the stars. If $\lambda$ be a unit-vector parallel to the polar axis, and $h$ the hour angle from the meridian, the operator is

$$\left(\cos\frac{h}{2} - \lambda\sin\frac{h}{2}\right)(\quad)\left(\cos\frac{h}{2} + \lambda\sin\frac{h}{2}\right)$$

or

$$L^{-1}(\quad)L$$

the inverse going first, because the *apparent* rotation is negative (clockwise).

If the upward line be $i$, and the southward $j$, we have

$$\lambda = i\sin l - j\cos l$$

where $l$ is the latitude of the observer. The meridian equatorial unit vector is

$$\mu = i\cos l + j\sin l$$

and $\lambda$, $\mu$, $k$ of course form a rectangular unit system.

The meridian unit-vector of a heavenly body is

$$\delta = i\cos(l-d) + j\sin(l-d)$$

$$= \lambda\sin d + \mu\cos d$$

where $d$ is its declination.

Hence when its hour-angle is $h$, its vector is

$$\delta' = L^{-1}\delta L$$

The vertical plane containing it intersects the horizon in

$$iVi\delta' = jSj\delta' + kSk\delta'$$

so that

$$\tan(azimuth) = \frac{Sk\delta'}{Sj\delta'} \tag{1}$$

[This may also be obtained directly from the last formula (1) of §114.]

To find its Amplitude, i.e. its azimuth at rising or setting, the hour-angle must be obtained from the condition

$$Si\delta' = 0 \tag{2}$$

These relations, with others immediately deducible from them, enable us (at once and for ever) to dispense with the hideous formulae of Spherical Trigonometry.

**122**. To show how readily they can be applied, let us translate the expressions above into the ordinary notation. This is effected at once by means of the expressions for $\lambda$, $\mu$, $L$, and $\delta$ above, which give by inspection

$$\delta' = \lambda\sin d + (\mu\cos h - k\sin h)\cos d$$

= x sin d + (fjb cos h k sin h) cos d, and we have from (1) and (2) of last section respectively

$$\tan(azimuth) = \frac{\sin h\cos d}{\cos l\sin d - \sin l\cos d\cos h} \tag{1}$$

$$\cos h + \tan l\tan d = 0 \tag{2}$$

In Capt. Weir s ingenious *Azimuth Diagram*, these equations are represented graphically by the rectangular coordinates of a system of confocal conics: viz.

$$\left.\begin{aligned} x &= \sin h\sec l \\ y &= \cos h\tan l \end{aligned}\right\} \tag{3}$$

The ellipses of this system depend upon $l$ alone, the hyperbolas upon $h$. Since (1) can, by means of (3), be written as

$$\tan(azimuth) = \frac{x}{\tan d - y}$$

we see that the azimuth can be constructed at once by joining with the point $0, -\tan d$, the intersection of the proper ellipse and hyperbola.

Equation (2) puts these expressions for the coordinates in the form

$$\left.\begin{array}{c} x = \sec l\sqrt{1-\tan^2 l\tan^2 d} \\ y = -\tan^2 l\tan d \end{array}\right\}$$

The elimination of $d$ gives the ellipse as before, but that of $l$ gives, instead of the hyperbolas, the circles

$$x^2+y^2-y(\tan d-\cot d)=1$$

The radius is

$$\frac{1}{2}(\tan d+\cot d)$$

and the coordinates of the centre are

$$0, \quad \frac{1}{2}(\tan d-\cot d)$$

123. A scalar equation in $\rho$, the vector of an undetermined point, is generally the equation of a *surface*; since we may use in it the expression

$$\rho = x\alpha$$

where $x$ is an unknown scalar, and $\alpha$ any assumed unit-vector. The result is an equation to determine $x$. Thus one or more points are found on the vector $x\alpha$, whose coordinates satisfy the equation; and the locus is a surface whose degree is determined by that of the equation which gives the values of $x$.

But a *vector* equation in $\rho$, as we have seen, generally leads to three scalar equations, from which the three rectangular or other components of the sought vector are to be derived. Such a vector equation, then, usually belongs to a definite number of *points* in space. But in certain cases these may form a *line*, and even a *surface*, the vector equation losing as it were one or two of the three scalar equations to which it is usually equivalent.

Thus while the equation

$$\alpha\rho = \beta$$

gives at once

$$\rho = \alpha^{-1}\beta$$

which is the vector of a definite point, since by making $\rho$ a *vector* we have evidently assumed

$$S\alpha\beta = 0$$

the closely allied equation

$$V\alpha\rho = \beta$$

is easily seen to involve

$$S\alpha\beta = 0$$

and to be satisfied by

$$\rho = \alpha^{-1}\beta + x\alpha$$

whatever be $x$. Hence the vector of any point whatever in the line drawn parallel to $\alpha$ from the extremity of $\alpha^{-1}\beta$ satisfies the given equation. [The difference between the results depends upon the fact that $S\alpha\rho$ is indeterminate in the second form, but definite ($= 0$) in the first.]

**124**. Again,

$$V\alpha\rho.V\rho\beta = (V\alpha\beta)^2$$

is equivalent to but two scalar equations. For it shows that $V\alpha\rho$ and $V\beta\rho$ are parallel, i.e. $\rho$ lies in the same plane as $\alpha$ and $\beta$, and can therefore be written (§24)

$$\rho = x\alpha + y\beta$$

where $x$ and $y$ are scalars as yet undetermined.

We have now

$$V\alpha\rho = yV\alpha\beta$$

$$V\rho\beta = xV\alpha\beta$$

which, by the given equation, lead to

$$xy = 1, \quad \text{or} \quad y = \frac{1}{x}$$

or finally

$$\rho = x\alpha + \frac{1}{x}\beta$$

which (§40) is the equation of a hyperbola whose asymptotes are in the directions of $\alpha$ and $\beta$.

**125**. Again, the equation

$$V.V\alpha\beta V\alpha\rho = 0$$

though apparently equivalent to three scalar equations, is really equivalent to one only. In fact we see by §91 that it may be written

$$-\alpha S.\alpha\beta\rho = 0$$

whence, if $\alpha$ be not zero, we have

$$S.\alpha\beta\rho = 0$$

and thus (§101) the only condition is that $\rho$ is coplanar with $\alpha$, $\beta$. Hence the equation represents the plane in which $\alpha$ and $\beta$ lie.

**126**. Some very curious results are obtained when we extend these processes of interpretation to functions of a *quaternion*

$$q = w + \rho$$

instead of functions of a mere *vector* $\rho$.

A scalar equation containing such a quaternion, along with quaternion constants, gives, as in last section, the equation of a surface, if we assign a definite value to $w$. Hence for successive values of $w$, we have successive surfaces belonging to a system ; and thus when $w$ is indeterminate the equation represents not a *surface*, as before, but a *volume*, in the sense that the vector of any point within that volume satisfies the equation.

Thus the equation

$$(Tq)^2 = a^2$$

or

$$w^2 - \rho^2 = a^2$$

or

$$(TP)^2 = a^2 - w^2$$

represents, for any assigned value of $w$, not greater than $a$, a sphere whose radius is $\sqrt{a^2 - w^2}$. Hence the equation is satisfied by the vector of any point whatever in the *volume* of a sphere of radius $a$, whose centre is origin.

Again, by the same kind of investigation,

$$(T(q - \beta))^2 = a^2$$

where $q = w + \rho$, is easily seen to represent the volume of a sphere of radius $a$ described about the extremity of $\beta$ as centre.

Also $S(q^2) = -a^2$ is the equation of infinite space less the space contained in a sphere of radius $a$ about the origin.

Similar consequences as to the interpretation of vector equations in quaternions may be readily deduced by the reader.

**127**. The following transformation is enuntiated without proof by Hamilton (*Lectures*, p. 587, and *Elements*, p. 299).

$$r^{-1}(r^2q^2)^{\frac{1}{2}}q^{-1} = U(rq + KrKq)$$

To prove it, let

$$r^{-1}(r^2q^2)^{\frac{1}{2}}q^{-1} = t$$

then

$$Tt = 1$$

and therefore

$$Kt = t^{-1}$$

But

$$(r^2q^2)^{\frac{1}{2}} = rtq$$

or

$$r^2q^2 = rtqrtq$$

or

$$rq = tqrt$$

Hence

$$KqKr = t^{-1}KrKqt^{-1}$$

or

$$KrKq = tKqKrt$$

Thus we have

$$U(rq \pm KrKq) = tU(qr \pm KqKr)t$$

or, if we put

$$s = U(qr \pm KqKr)$$

$$Ks = \pm tst$$

Hence

$$sKs = (Ts)^2 = 1 = \pm stst$$

which, if we take the positive sign, requires

$$st = \pm 1$$

or

$$t = \pm s^{-1} = \pm UKs$$

which is the required transformation.

[It is to be noticed that there are other results which might have been arrived at by using the negative sign above ; some involving an arbitrary unit-vector, others involving the imaginary of ordinary algebra.]

**128**. As a final example, we take a transformation of Hamilton's, of great importance in the theory of surfaces of the second order.

Transform the expression

$$(S\alpha\rho)^2 + (S\beta\rho)^2 + (S\gamma\rho)^2$$

in which $\alpha$, $\beta$, $\gamma$ are any three mutually rectangular vectors, into the form

$$\left(\frac{T(\iota\rho + \rho\kappa)}{\kappa^2 - \iota^2}\right)^2$$

which involves only two vector-constants, $\iota$, $\kappa$.

[The student should remark here that $\iota$, $\kappa$, two undetermined vectors, involve six disposable constants : and that $\alpha$, $\beta$, $\gamma$, being a *rectangular* system, involve also only six constants.]

$$\begin{aligned} \{T(\iota\rho + \rho\kappa)\}^2 &= (\iota\rho + \rho\kappa)(\rho\iota + \kappa\rho) \quad (\S\S 52, 55) \\ &= (\iota^2 + \kappa^2)\rho^2 + (\iota\rho\kappa\rho + \rho\kappa\rho\iota) \\ &= (\iota^2 + \kappa^2)\rho^2 + 2S.\iota\rho\kappa\rho \\ &= (\iota - \kappa)^2\rho^2 + 4S\iota\rho S\kappa\rho \end{aligned}$$

Hence

$$(S\alpha\rho)^2 + (S\beta\rho)^2 + (S\gamma\rho)^2 = \frac{(\iota - \kappa)^2}{(\kappa^2 - \iota^2)^2}\rho^2 + 4\frac{S\iota\rho S\kappa\rho}{(\kappa^2 - \iota^2)^2}$$

But

$$\alpha^{-2}(S\alpha\rho)^2 + \beta^{-2}(S\beta\rho)^2 + \gamma^{-2}(S\gamma\rho)^2 = \rho^2 \quad (\S\S 25, 73).$$

Multiply by $\beta^2$ and subtract, we get

$$\left(1 - \frac{\beta^2}{\alpha^2}\right)(S\alpha\rho)^2 - \left(\frac{\beta^2}{\gamma^2} - 1\right)(S\gamma\rho)^2 = \left\{\frac{(\iota - \kappa)^2}{(\kappa^2 - \iota^2)^2} - \beta^2\right\}\rho^2 + 4\frac{S\iota\rho S\kappa\rho}{(\kappa^2 - \iota^2)^2}$$

The left side breaks up into two real factors if $\beta^2$ be intermediate in value to $\alpha^2$ and $\gamma^2$: and that the right side may do so the term in $\rho^2$ must vanish. This condition gives

$$\beta^2 = \frac{(\iota - \kappa)^2}{(\kappa^2 - \iota^2)^2}$$

and the identity becomes

$$S\left\{\alpha\sqrt{\left(1 - \frac{\beta^2}{\alpha^2}\right)} + \gamma\sqrt{\left(\frac{\beta^2}{\gamma^2} - 1\right)}\right\}\rho S\left\{\alpha\sqrt{\left(1 - \frac{\beta^2}{\alpha^2}\right)} - \gamma\sqrt{\left(\frac{\beta^2}{\gamma^2} - 1\right)}\right\}\rho = 4\frac{S\iota\rho S\kappa\rho}{(\kappa^2 - \iota^2)^2}$$

Hence we must have

$$\frac{2\iota}{\kappa^2-\iota^2} = p\left\{\alpha\sqrt{\left(1-\frac{\beta^2}{\alpha^2}\right)}+\gamma\sqrt{\left(\frac{\beta^2}{\gamma^2}-1\right)}\right\}$$

$$\frac{2\kappa}{\kappa^2-\iota^2} = \frac{1}{p}\left\{\alpha\sqrt{\left(1-\frac{\beta^2}{\alpha^2}\right)}-\gamma\sqrt{\left(\frac{\beta^2}{\gamma^2}-1\right)}\right\}$$

where $\rho$ is an undetermined scalar.

To determine $\rho$, substitute in the expression for $\beta^2$, and we find

$$\begin{aligned} 4\beta^2 = \tfrac{4(\iota-\kappa)^2}{(\kappa^2-\iota^2)^2} &= \left(p-\tfrac{1}{p}\right)^2(\alpha^2-\beta^2)+\left(p+\tfrac{1}{p}\right)^2(\beta^2-\gamma^2) \\ &= \left(p^2+\tfrac{1}{p^2}\right)(\alpha^2-\gamma^2)-2(\alpha^2+\gamma^2)+4\beta^2 \end{aligned}$$

Thus the transformation succeeds if

$$p^2+\frac{1}{p^2} = \frac{2(\alpha^2+\gamma^2)}{\alpha^2-\gamma^2}$$

which gives

$$p+\frac{1}{p} = \pm 2\sqrt{\frac{\alpha^2}{\alpha^2-\gamma^2}}$$

$$p-\frac{1}{p} = \pm 2\sqrt{\frac{\gamma^2}{\alpha^2-\gamma^2}}$$

Hence

$$\frac{4(\kappa^2-\iota^2)}{(\kappa^2-\iota^2)^2} = \left(\frac{1}{p^2}-p^2\right)(\alpha^2-\gamma^2) = \pm 4\sqrt{\alpha^2\gamma^2}$$

$$(\kappa^2-\iota^2)^{-1} = \pm T\alpha T\gamma$$

Again

$$p = \frac{T\alpha+T\gamma}{\sqrt{\gamma^2-\alpha^2}}, \quad \frac{1}{p} = \frac{T\alpha-T\gamma}{\sqrt{\gamma^2-\alpha^2}}$$

and therefore

$$2\iota = \frac{T\alpha+T\gamma}{T\alpha T\gamma}\left(\sqrt{\frac{\beta^2-\alpha^2}{\gamma^2-\alpha^2}}U\alpha+\sqrt{\frac{\gamma^2-\beta^2}{\gamma^2-\alpha^2}}U\gamma\right)$$

$$2\kappa = \frac{T\alpha-T\gamma}{T\alpha T\gamma}\left(\sqrt{\frac{\beta^2-\alpha^2}{\gamma^2-\alpha^2}}U\alpha-\sqrt{\frac{\gamma^2-\beta^2}{\gamma^2-\alpha^2}}U\gamma\right)$$

Thus we have proved the possibility of the transformation, and determined the transforming vectors $\iota$, $\kappa$.

**129**. By differentiating the equation

$$(S\alpha\rho)^2+(S\beta\rho)^2+(S\gamma\rho)^2 = \left(\frac{T(\iota\rho+\rho\kappa)}{(\kappa^2-\iota^2)}\right)^2$$

we obtain, as will be seen in Chapter IV, the following,

$$S\alpha\rho S\alpha\rho' + S\beta\rho S\beta\rho' + S\gamma\rho S\gamma\rho' = \frac{S.(\iota\rho + \rho\kappa)(\kappa\rho' + \rho'\iota)}{(\kappa^2 - \iota^2)^2}$$

where $\rho$ also may be any vector whatever.

This is another very important formula of transformation ; and it will be a good exercise for the student to prove its truth by processes analogous to those in last section. We may merely observe, what indeed is obvious, that by putting $\rho' = \rho$ it becomes the formula of last section. And we see that we may write, with the recent values of $\iota$ and $\kappa$ in terms of $\alpha$, $\beta$, $\gamma$, the identity

$$\begin{aligned} \alpha S\alpha\rho + \beta S\beta\rho + \gamma S\gamma\rho &= \frac{(\iota^2 + \kappa^2)\rho + 2V.\iota\rho\kappa}{(\kappa^2 - \iota^2)^2} \\ &= \frac{(\iota - \kappa)^2\rho + 2(\iota S\kappa\rho + \kappa S\iota\rho)}{(\kappa^2 - \iota^2)^2} \end{aligned}$$

**130**. In various quaternion investigations, especially in such as involve *imaginary* intersections of curves and surfaces, the old imaginary of algebra of course appears. But it is to be particularly noticed that this expression is analogous to a scalar and not to a vector, and that like real scalars it is commutative in multiplication with all other factors. Thus it appears, by the same proof as in algebra, that any quaternion expression which contains this imaginary can always be broken up into the sum of two parts, one real, the other multiplied by the first power of $\sqrt{-1}$. Such an expression, viz.

$$q = q' + \sqrt{-1}q''$$

where $q'$ and $q''$ are real quaternions, is called by Hamilton a BIQUATERNION. [The student should be warned that the term Biquaternion has since been employed by other writers in the sense sometimes of a "set" of 8 elements, analogous to the Quaternion 4 ; sometimes for an expression $q' + \theta q''$ where $\theta$ is not the algebraic imaginary. By them Hamilton s Biquaternion is called simply a quaternion with non-real constituents.] Some little care is requisite in the management of these expressions, but there is no new difficulty. The points to be observed are: first, that any biquaternion can be divided into a real and an imaginary part, the latter being the product of $\sqrt{-1}$ by a real quaternion; second, that this $\sqrt{-1}$ is commutative with all other quantities in multiplication; third, that if two biquaternions be equal, as

$$q' + \sqrt{-1}\, q'' = r' + \sqrt{-1}\, r''$$

we have, as in algebra,

$$q' = r', \quad q'' = r''$$

so that an equation between biquaternions involves in general *eight* equations between scalars. Compare §80.

**131**. We have obviously, since $\sqrt{-1}$ is a scalar,

$$S(q' + \sqrt{-1}\, q'') = Sq' + \sqrt{-1}\, Sq''$$

$$V(q' + \sqrt{-1}\, q'') = Vq' + \sqrt{-1}\, Vq''$$

Hence (§103)

$$\{T(q' + \sqrt{-1}\, q'')\}^2$$

$$= (Sq' + \sqrt{-1}\, Sq'' + Vq' + \sqrt{-1}\, Vq'')(Sq' + \sqrt{-1}\, Sq'' - Vq' - \sqrt{-1}\, Vq'')$$
$$= (Sq' + \sqrt{-1}\, Sq'')^2 - (Vq' + \sqrt{-1}\, Vq'')^2$$
$$= (Tq')^2 - (Tq'')^2 + 2\sqrt{-1}\, S.q'Kq''$$

The only remark which need be made on such formulae is this, that *the tensor of a biquaternion may vanish while both of the component quaternions are finite.*

Thus, if

$$Tq' = Tq''$$

and

$$S.q'Kq'' = 0$$

the above formula gives

$$T(q' + \sqrt{-1}\, q'') = 0$$

The condition

$$S.q'Kq'' = 0$$

may be written

$$Kq'' = q^{'-1}\alpha, \quad \text{or} \quad q'' = -\alpha K q^{'-1} = -\frac{\alpha q'}{(Tq')^2}$$

where $\alpha$ is any vector whatever.

Hence

$$Tq' = Tq'' = TKq'' = \frac{T\alpha}{Tq''}$$

and therefore

$$Tq'(Uq' - \sqrt{-1}\, U\alpha.Uq') = (1 - \sqrt{-1}\, U\alpha)q'$$

is the general form of a biquaternion whose tensor is zero.

**132**. More generally we have, $q$, $r$, $q'$, $r'$ being any four real and non-evanescent quaternions,

$$(q + \sqrt{-1}\, q')(r + \sqrt{-1}\, r') = qr - q'r' + \sqrt{-1}\, (qr' + q'r)$$

That this product may vanish we must have

$$qr = q'r'$$

and

$$qr' = -q'r$$

Eliminating $r'$ we have

$$qq^{'-1}qr = -q'r$$

which gives

$$(q^{'-1}q)^2 = -1$$

i.e.

$$q = q'\alpha$$

where $\alpha$ is some unit-vector.

And the two equations now agree in giving

$$-r = \alpha r'$$

so that we have the biquaternion factors in the form

$$q'(\alpha + \sqrt{-1}) \quad \text{and} \quad -(\alpha - \sqrt{-1})r'$$

and their product is

$$-q'(\alpha + \sqrt{-1})(\alpha - \sqrt{-1})r'$$

which, of course, vanishes.

[A somewhat simpler investigation of the same proposition may be obtained by writing the biquaternions as

$$q'(q'^{-1}q + \sqrt{-1}) \quad \text{and} \quad (rr'^{-1} + \sqrt{-1})r'$$

or

$$q'(q'' + \sqrt{-1}) \quad \text{and} \quad (r'' + \sqrt{-1})r'$$

and showing that

$$q'' = -r'' = \alpha \quad \text{where } T\alpha = 1]$$

From this it appears that if the product of two *bivectors*

$$\rho + \sigma\sqrt{-1} \quad \text{and} \quad \rho' + \sigma'\sqrt{-1}$$

is zero, we must have

$$\sigma^{-1}\rho = -\rho'\sigma'^{-1} = U\alpha$$

where $\alpha$ may be any vector whatever. But this result is still more easily obtained by means of a direct process.

**133**. It may be well to observe here (as we intend to avail our selves of them in the succeeding Chapters) that certain abbreviated forms of expression may be used when they are not liable to confuse, or lead to error. Thus we may write

$$T^2q \quad \text{for} \quad (Tq)^2$$

just as we write

$$\cos^2\theta \quad \text{for} \quad (\cos\theta)^2$$

although the true meanings of these expressions are

$$T(Tq) \quad \text{and} \quad \cos(\cos\theta)$$

The former is justifiable, as $T(Tq) = Tq$, and therefore $T^2q$ is not required to signify the second tensor (or tensor of the tensor) of $q$. But the trigonometrical usage is defensible only on the score of convenience, and is habitually violated by the employment of $cos^{-1}x$ in its natural and proper sense. Similarly we may write

$$S^2q \quad \text{for} \quad (Sq)^2, \quad \&\text{c.}$$

but it may be advisable not to use

$$Sq^2$$

as the equivalent of either of those just written; inasmuch as it might be confounded with the (generally) different quantity

$$S.q^2 \quad \text{or} \quad S(q^2)$$

although this is rarely written without the point or the brackets.

The question of the use of points or brackets is one on which no very definite rules can be laid down. A beginner ought to use them freely, and he will soon learn by trial which of them are absolutely necessary to prevent ambiguity.

In the present work this course has been adopted:– the earlier examples in each part of the subject being treated with a free use of points and brackets, while in the later examples superfluous marks of the kind are gradually got rid of.

It may be well to indicate some general principles which regulate the omission of these marks. Thus in $S.\alpha\beta$ or $V.\alpha\beta$ the point is obviously unnecessary:– because $S\alpha = 0$, and $V\alpha = \alpha$ so that the $S$ would annihilate the term if it applied to $\alpha$ alone, while in the same case the $V$ would be superfluous. But in $S.qr$ and $V.qr$, the point (or an equivalent) is indispensable, for $Sq.r$, and $Vq.r$ are usually quite different from the first written quantities. In the case of $K$, and of $d$ (used for scalar differentiation), the *omission* of the point indicates that the operator acts *only* on the nearest factor:– thus

$$Kqr = (Kq)r = Kq.r, \quad dqr = (dq)r = dq.r$$

Kqr = (Kq) r = Kq . r, dqr = (dq) r=dq.r; while, if its action extend farther, we write

$$K.qr = K(qr), \quad d.qr = d(qr) \quad \&c.$$

In more complex cases we must be ruled by the general principle of dropping nothing which is essential. Thus, for instance

$$V(pK(dq)V(Vq.r))$$

may be written without ambiguity as

$$V(pK(dq)V(Vq.r))$$

but nothing more can be dropped without altering its value.

Another peculiarity of notation, which will occasionally be required, shows *which portions* of a complex product are affected by an operator. Thus we write

$$\nabla S\sigma\tau$$

if $\nabla$ operates on $\sigma$ and also on $\tau$, but

$$\nabla_1 S\sigma\tau_1$$

if it operates on $\tau$ alone. See, in this connection, the last Example at the end of Chap. IV. below.

**134**. The beginner may expect to be at first a little puzzled with this aspect of the notation; but, as he learns more of the subject, he will soon see clearly the distinction between such an expression as

$$S.V\alpha\beta V\beta\gamma$$

where we may omit at pleasure either the point or the first V without altering the value, and the very different one

$$S\alpha\beta.V\beta\gamma$$

which admits of no such changes, without alteration of its value.

All these simplifications of notation are, in fact, merely examples of the transformations of quaternion expressions to which part of this Chapter has been devoted. Thus, to take a very simple ex ample, we easily see that

$$\begin{aligned} S.V\alpha\beta V\beta\gamma &= SV\alpha\beta V\beta\gamma = S.\alpha\beta V\beta\gamma = S\alpha V.\beta V\beta\gamma = -S\alpha V.(V\beta\gamma)\beta \\ &= S\alpha V.(V\gamma\beta)\beta = S.\alpha V(\gamma\beta)\beta = S.V(\gamma\beta)\beta\alpha = SV\gamma\beta V\beta\alpha \\ &= S.\gamma\beta V\beta\alpha = S.K(\beta\gamma)V\beta\alpha = S.\beta\gamma KV\beta\alpha = -S.\beta\gamma V\beta\alpha \\ &= S.V\gamma\beta V\beta\alpha, \&\text{c., \&c.} \end{aligned}$$

The above group does not nearly exhaust the list of even the simpler ways of expressing the given quantity. We recommend it to the careful study of the reader. He will find it advisable, at first, to use stops and brackets pretty freely; but will gradually learn to dispense with those which are not absolutely necessary to prevent ambiguity.

There is, however, one additional point of notation to which the reader s attention should be most carefully directed. A very simple instance will suffice. Take the expressions

$$\frac{\beta}{\gamma}.\frac{\gamma}{\alpha} \quad \text{and} \quad \frac{\beta\gamma}{\gamma\alpha}$$

The first of these is

$$\beta\gamma^{-1}.\gamma\alpha^{-1} = \beta\alpha^{-1}$$

and presents no difficulty. But the second, though at first sight it closely resembles the first, is in general totally different in value, being in fact equal to

$$\beta\gamma\alpha^{-1}\gamma^{-1}$$

For the denominator must be treated as *one quaternion*. If, then, we write

$$\frac{\beta\gamma}{\gamma\alpha} = q$$

we have

$$\beta\gamma = q\gamma\alpha$$

so that, as stated above,

$$q = \beta\gamma\alpha^{-1}\gamma^{-1}$$

We see therefore that

$$\frac{\beta}{\gamma}.\frac{\gamma}{\alpha} = \frac{\beta}{\alpha} = \frac{\beta\gamma}{\alpha\gamma}; \quad \text{but } \textit{not} \quad = \frac{\beta\gamma}{\gamma\alpha}$$

## 4.7 Examples to Chapter 3

**1**. Investigate, by quaternions, the requisite formulae for changing from any one set of coordinate axes to another ; and derive from your general result, and also from special investiga tions, the usual expressions for the following cases:

(a) Rectangular axes turned about z through any angle.

(b) Rectangular axes turned into any new position by rota tion about a line equally inclined to the three.

(c) Rectangular turned to oblique, one of the new axes lying in each of the former coordinate planes.

**2**. Point out the distinction between

$$\left(\frac{\alpha+\beta}{\alpha}\right)^2 \quad \text{and} \quad \frac{(\alpha+\beta)^2}{\alpha^2}$$

and find the value of their difference.

If

$$T\beta/\alpha = 1 \quad \text{and} \quad U\frac{\alpha+\beta}{\alpha} = \left(\frac{\beta}{\alpha}\right)^{\frac{1}{2}}$$

Show also that

$$\frac{\alpha+\beta}{\alpha-\beta} = \frac{V\alpha\beta}{1+S\alpha\beta'}$$

and

$$\frac{\alpha-\beta}{\alpha+\beta} = -\frac{V\alpha\beta}{1-S\alpha\beta'}$$

provided $\alpha$ and $\beta$ be unit-vectors. If these conditions are not fulfilled, what are the true values ?

**3**. Show that, whatever quaternion $r$ may be, the expression

$$\alpha r + r\beta$$

in which $\alpha$ and $\beta$ are any two unit- vectors, is reducible to the form

$$l(\alpha+\beta) + m(\alpha\beta - 1)$$

where $l$ and $m$ are scalars.

**4**. If $Tp = T\alpha = T\beta = 1$, and $S.\alpha\beta\rho = 0$ show by direct transformations that

$$S.U(\rho-\alpha)U(\rho-\beta) = \pm\sqrt{\frac{1}{2}(1-S\alpha\beta)}$$

Interpret this theorem geometrically.

**5**. If $S\alpha\beta = 0$, $T\alpha = T\beta = 1$, show that

$$(1+\alpha^m)\beta = 2\cos\frac{m\pi}{4}\alpha^{\frac{m}{2}}\beta = 2S\alpha^{\frac{m}{2}}.\alpha^{\frac{m}{2}}\beta$$

**6**. Put in its simplest form the equation

$$\rho S.V\alpha\beta V\beta\gamma V\gamma\alpha = aV.V\gamma\alpha V\alpha\beta + bV.V\alpha\beta V\beta\gamma + cV.V\beta\gamma V\gamma\alpha$$

and show that

$$a = S.\beta\gamma\rho, \quad \&\text{c.}$$

**7**. Show that any quaternion may in general, in one way only, be expressed as a homogeneous linear function of four given quaternions. Point out the nature of the exceptional cases. Also find the simplest form in which any quaternion may generally be expressed in terms of two given quaternions.

**8**. Prove the following theorems, and exhibit them as properties of determinants :

(a) $S.(\alpha+\beta)(\beta+\gamma)(\gamma+\alpha) = 2S.\alpha\beta\gamma$

(b) $S.V\alpha\beta V\beta\gamma V\gamma\alpha = -(S.\alpha\beta\gamma)^2$

(c) $S.V(\alpha+\beta)(\beta+\gamma)V(\beta+\gamma)(\gamma+\alpha)V(\gamma+\alpha)(\alpha+\beta) = -4(S.\alpha\beta\gamma)^2$

(d) $S.V(V\alpha\beta V\beta\gamma)V(V\beta\gamma V\gamma\alpha)V(V\gamma\alpha V\alpha\beta) = -(S.\alpha\beta\gamma)^4$

(e) $S.\delta\epsilon\zeta = -16(S.\alpha\beta\gamma)^4$
where

$$\delta = V(V(\alpha+\beta)(\beta+\gamma)V(\beta+\gamma)(\gamma+\alpha))$$
$$\epsilon = V(V(\beta+\gamma)(\gamma+\alpha)V(\gamma+\alpha)(\alpha+\beta))$$
$$\zeta = V(V(\gamma+\alpha)(\alpha+\beta)V(\alpha+\beta)(\beta+\gamma))$$

**9**. Prove the common formula for the product of two determinants of the third order in the form

$$S.\alpha\beta\gamma S.\alpha_1\beta_1\gamma_1 = \begin{vmatrix} S\alpha\alpha_1 & S\beta\alpha_1 & S\gamma\alpha_1 \\ S\alpha\beta_1 & S\beta\beta_1 & S\gamma\beta_1 \\ S\alpha\gamma_1 & S\beta\gamma_1 & S\gamma\gamma_1 \end{vmatrix}$$

**10**. Show that, whatever be the eight vectors involved,

$$\begin{vmatrix} S\alpha\alpha_1 & S\alpha\beta_1 & S\alpha\gamma_1 & S\alpha\delta_1 \\ S\beta\alpha_1 & S\beta\beta_1 & S\beta\gamma_1 & S\beta\delta_1 \\ S\gamma\alpha_1 & S\gamma\beta_1 & S\gamma\gamma_1 & S\gamma\delta_1 \\ S\delta\alpha_1 & S\delta\beta_1 & S\delta\gamma_1 & S\delta\delta_1 \end{vmatrix} = S.\alpha\beta\gamma S.\beta_1\gamma_1\delta_1 S\alpha_1(\delta-\delta) = 0$$

If the single term $S\alpha\alpha_1$, be changed to $S\alpha_0\alpha_1$, the value of the determinant is

$$S.\beta\gamma\delta S.\beta_1\gamma_1\delta_1 S\alpha_1(\alpha_0-\alpha)$$

State these as propositions in spherical trigonometry.

Form the corresponding null determinant for any two groups of five quaternions : and give its geometrical interpretation.

**11**. If, in §102, $\alpha$, $\beta$, $\gamma$ be three mutually perpendicular vectors, can anything be predicated as to $\alpha_1$, $\beta_1$, $\gamma_1$? If $\alpha$, $\beta$, $\gamma$ be rectangular unit-vectors, what of $\alpha_1$, $\beta_1$, $\gamma_1$?

**12**. If $\alpha$, $\beta$, $\gamma$, $\alpha'$, $\beta'$, $\gamma'$ be two sets of rectangular unit-vectors, show that

$$S\alpha\alpha' = S\gamma\beta' S\beta\gamma' = S\beta\beta' S\gamma\gamma' \quad \&c.\ \&c.$$

**13**. The lines bisecting pairs of opposite sides of a quadrilateral (plane or gauche) are perpendicular to each other when the diagonals of the quadrilateral are equal.

**14**. Show that

(a) $S.q^2 = 2S^2q - T^2q$

(b) $S.q^3 = S^3q - 3SqT^2Vq$

(c) $\alpha^2\beta^2\gamma^2 + S^2.\alpha\beta\gamma = V^2.\alpha\beta\gamma$

(d) $S(V.\alpha\beta\gamma V.\beta\gamma\alpha V.\gamma\alpha\beta) = 4S\alpha\beta S\beta\gamma S\gamma\alpha S.\alpha\beta\gamma$

(e) $V.q^3 = (2S^2q - T^2Vq)Vq$

(f) $qUVq^{-1} = -Sq.UVq + TVq$

and interpret each as a formula in plane or spherical trigonometry.

**15**. If $q$ be an undetermined quaternion, what loci are represented by

(a) $(q\alpha^{-1})^2 = -a^2$

(b) $(q\alpha^{-1})^4 = a^4$

(c) $S.(q-\alpha)^2 = a^2$

where $a$ is any given scalar and $\alpha$ any given vector ?

**16**. If $q$ be any quaternion, show that the equation

$$Q^2 = q^2$$

is satisfied, not alone by $Q = \pm q$, but also by

$$Q = \pm\sqrt{-1}(Sq.UVq - TVq)$$

(Hamilton, *Lectures*, p. 673.)

**17**. Wherein consists the difference between the two equations

$$T^2\frac{\rho}{\alpha} = 1 \quad \text{and} \quad \left(\frac{\rho}{\alpha}\right)^2 = -1$$

What is the full interpretation of each, $\alpha$ being a given, and p an undetermined, vector?

**18**. Find the *full* consequences of each of the following groups of equations, as regards both the unknown vector $\rho$ and the given vectors $\alpha$, $\beta$, $\gamma$:

$$(a)\quad \begin{matrix} S.\alpha\beta\rho = 0 \\ S.\beta\gamma\rho = 0 \end{matrix} \qquad (b)\quad \begin{matrix} S\alpha\rho = 0 \\ S.\alpha\beta\rho = 0 \\ S\beta\rho = 0 \end{matrix} \qquad (c)\quad \begin{matrix} S\alpha\rho = 0 \\ S.\alpha\beta\rho = 0 \\ S.\alpha\beta\gamma\rho = 0 \end{matrix}$$

**19**. From §§74, 110, show that, if $\epsilon$ be any unit-vector, and $m$ any scalar,

$$\epsilon^m = \cos\frac{m\pi}{2} + \epsilon\sin\frac{m\pi}{2}$$

Hence show that if $\alpha$, $\beta$, $\gamma$ be radii drawn to the corners of a triangle on the unit-sphere, whose spherical excess is $m$ right angles,

$$\frac{\alpha+\beta}{\beta+\gamma}.\frac{\gamma+\alpha}{\alpha+\beta}.\frac{\beta+\gamma}{\gamma+\alpha} = \alpha^m$$

Also that, if $A$, $B$, $C$ be the angles of the triangle, we have

$$\gamma^{\frac{2C}{\pi}}\beta^{\frac{2B}{\pi}}\alpha^{\frac{2A}{\pi}} = -1$$

**20**. Show that for any three vectors $\alpha$, $\beta$, $\gamma$ we have

$$(U\alpha\beta)^2 + (U\beta\gamma)^2 + (U\alpha\gamma)^2 + (U.\alpha\beta\gamma)^2 + 4U\alpha\gamma.SU\alpha\beta SU\beta\gamma = -2$$

(Hamilton, *Elements*, p. 388.)

**21**. If $a_1$, $a_2$, $a_3$, $x$ be any four scalars, and $\rho_1$, $\rho_2$, $\rho_3$ any three vectors, show that

$$(S.\rho_1\rho_2\rho_3)^2 + (\sum .a_1V\rho_2\rho_3)^2 + x^2(\sum V\rho_1\rho_2)^2 -$$

$$x^2(\sum .a_1(\rho_2 - \rho_3))^2 + 2\prod(x^2 + S\rho_1\rho_2 + a_1a_2)$$

$$= 2\prod(x^2 + \rho^2) + 2\prod a^2 +$$

$$\sum\{(x^2 + a_1^2 + \rho_1^2)((V\rho_2\rho_3)^2 + 2a_2a_3(x^2 + S\rho_2\rho_3) - x^2(\rho_2 - \rho_3)^2)\}$$

where $\prod a^2 = a_1^2a_2^2a_3^2$

Verify this formula by a simple process in the particular case

$$a_1 = a_2 = a_3 = x = 0$$

(*Ibid*)

**22**. Eliminate $p$ from the equations

$$V.\beta\rho\alpha\rho = 0, \quad S\gamma\rho = 0$$

and state the problem and its solution in a geometrical form.

**23**. If $p$, $q$, $r$, $s$ be four versors, such that

$$qp = -sr = \alpha$$

$$rq = -ps = \beta$$

where $\alpha$ and $\beta$ are unit-vectors; show that

$$S(V.VsVqV.VrVp) = 0$$

Interpret this as a property of a spherical quadrilateral.

**24**. Show that, if $pq$, $rs$, $pr$, and $qs$ be vectors, we have

$$S(V.VpVsV.VqVr) = 0$$

**25**. If $\alpha$, $\beta$, $\gamma$ be unit-vectors,

$$V\beta\gamma S.\alpha\beta\gamma = -\alpha(1 - S^2\beta\gamma) - \beta(S\alpha\gamma S\beta r + S\alpha\beta) - \gamma(S\alpha\beta S\beta\gamma + S\alpha\gamma)$$

**26**. If $i$, $j$, $k$, $i'$, $j'$, $k'$, be two sets of rectangular unit-vectors, show that

$$\begin{aligned} S.Vii'Vjj'Vkk' &= (Sij')^2 - (Sji')^2 \\ &= (Sjk')^2 - (Skj')^2 = \&\text{c.} \end{aligned}$$

and find the values of the vector of the same product.

**27**. If $\alpha$, $\beta$, $\gamma$ be a rectangular unit-vector system, show that, whatever be $\lambda$, $\mu$, $\nu$

$$\lambda S^2i\alpha + \mu S^2j\gamma + \nu S^2k\beta$$

$$\lambda S^2k\gamma + \mu S^2i\beta + \nu S^2j\alpha$$

and

$$\lambda S^2j\beta + \mu S^2k\alpha + \nu S^2i\gamma$$

are coplanar vectors. What is the connection between this and the result of the preceding example ?

## 4.8 Axiom Examples

The basic operation for creating quaternions is **quatern**. This is a quaternion over the rational numbers.

```
q:=quatern(2/11,-8,3/4,1)
```

$$\frac{2}{11} - 8\ i + \frac{3}{4}\ j + k$$

```
                                   Type:  Quaternion Fraction Integer
```

This is a quaternion over the integers.

```
r:=quatern(1,2,3,4)
```

$$1 + 2\ i + 3\ j + 4\ k$$

```
                                   Type:  Quaternion Integer
```

We can also construct quaternions with complex components. First we construct a complex number.

```
b:=complex(3,4)
```

$$3 + 4\ i$$

```
                                   Type:  Complex Integer
```

and then we use it as a component in a quaternion.

```
s:=quatern(3,1/7,b,2)
```

$$3 + \frac{1}{7}\ i + (3 + 4\ i)\ j + 2\ k$$

```
                          Type:  Quaternion Complex Fraction Integer
```

Notice that the $i$ component of the complex number has no relation to the $i$ component of the quaternion even though they use the same symbol by convention.

The four parts of a quaternion are the real part, the $i$ imaginary part, the $j$ imaginary part, and the $k$ imaginary part. The **real** function returns the real part.

```
real q
```

$$\frac{2}{11}$$

```
                                   Type:  Fraction Integer
```

The **imagI** function returns the $i$ imaginary part.

```
imagI q
```

$$-8$$

```
                                   Type:  Fraction Integer
```

The **imagJ** function returns the $j$ imaginary part.

```
imagJ q
```

$$\frac{3}{4}$$

```
                                   Type:  Fraction Integer
```

The **imagK** function returns the $k$ imaginary part.

```
imagK q
                                    1
                                                    Type:  Fraction Integer
```

Quaternions satisfy a very fundamental relationship between the parts, namely that

$$i^2 = j^2 = k^2 = ijk = -1$$

. This is similar to the requirement in complex numbers of the form $a + bi$ that $i^2 = -1$.

The set of quaternions is denoted by $\mathbb{H}$, whereas the integers are denoted by $\mathbb{Z}$ and the complex numbers by $\mathbb{C}$.

Quaternions are not commutative which means that in general

$$AB \neq BA$$

for any two quaternions, A and B. So, for instance,

```
q*r
                  437     84          1553     523
                  ---  - --- i + ---- j - --- k
                   44     11        44       22
                                    Type:  Quaternion Fraction Integer
```

```
r*q
                  437     84          1439     599
                  ---  - --- i - ---- j + --- k
                   44     11        44       22
                                    Type:  Quaternion Fraction Integer
```

and these are clearly not equal.

Complex $2 \times 2$ matrices form an alternate, equivalent representation of quaternions. These matrices have the form:

$$\begin{bmatrix} u & v \\ -\overline{v} & \overline{u} \end{bmatrix}$$

=

$$\begin{bmatrix} a+bi & c+di \\ -c+di & a-bi \end{bmatrix}$$

where $u$ and $v$ are complex, $\overline{u}$ is complex conjugate of $u$, $\overline{z}$ is the complex conjugate of $z$, and a,b,c, and d are real.

Within the quaternion each component operator represents a basis element in $\mathbb{R}^4$ thus:

$$1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$i = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

# Chapter 5

# Clifford Algebra

This is quoted from John Fletcher's web page [Flet09] (with permission).

The theory of Clifford Algebra includes a statement that each Clifford Algebra is isomorphic to a matrix representation. Several authors discuss this and in particular Ablamowicz [Abla98] gives examples of derivation of the matrix representation. A matrix will itself satisfy the characteristic polynomial equation obeyed by its own eigenvalues. This relationship can be used to calculate the inverse of a matrix from powers of the matrix itself. It is demonstrated that the matrix basis of a Clifford number can be used to calculate the inverse of a Clifford number using the characteristic equation of the matrix and powers of the Clifford number. Examples are given for the algebras Clifford(2), Clifford(3) and Clifford(2,2).

## 5.1 Introduction

Introductory texts on Clifford algebra state that for any chosen Clifford Algebra there is a matrix representation which is equivalent. Several authors discuss this in more detail and in particular, Ablamowicz [Abla98] shows that the matrices can be derived for each algebra from a choice of idempotent, a member of the algebra which when squared gives itself. The idea of this paper is that any matrix obeys the characteristic equation of its own eigenvalues, and that therefore the equivalent Clifford number will also obey the same characteristic equation. This relationship can be exploited to calculate the inverse of a Clifford number. This result can be used symbolically to find the general form of the inverse in a particular algebra, and also in numerical work to calculate the inverse of a particular member. This latter approach needs the knowledge of the matrices. Ablamowicz has provided a method for generating them in the form of a Maple implementation. This knowledge is not believed to be new, but the theory is distributed in the literature and the purpose of this paper is to make it clear. The examples have been first developed using a system of symbolic algebra described in another paper by this author [Flet01].

## 5.2 Clifford Basis Matrix Theory

The theory of the matrix basis is discussed extensively by Ablamowicz. This theory will be illustrated here following the notation of Ablamowicz by reference to Clifford(2) algebra and

can be applied to other Clifford Algebras. For most Clifford algebras there is at least one primitive idempotent, such that it squares to itself. For Clifford (2), which has two basis members $e_1$ and $e_2$, one such idempotent involves only one of the basis members, $e_1$, i.e.

$$f_1 = f = \frac{1}{2}(1 + e_1)$$

If the idempotent is mutiplied by the other basis function $e_2$, other functions can be generated:

$$f_2 = e_2 f = \left(\frac{1}{2} - \frac{1}{2}e_1\right) e_2$$

$$f_3 = f e_2 = \left(\frac{1}{2} + \frac{1}{2}e_1\right) e_2$$

$$f_4 = e_2 f e_2 = \frac{1}{2} - \frac{1}{2}e_1$$

Note that $fe_2 2f = 0$. These four functions provide a means of representing any member of the space, so that if a general member c is given in terms of the basis members of the algebra

$$c = a_0 + a_1 e_1 + a_2 e_2 + a_3 e_1 e_2$$

it can also be represented by a series of terms in the idempotent and the other functions.

$$\begin{aligned} c &= a_{11} f_1 + a_{21} f_2 + a_{12} f_3 + a_{22} f_4 \\ &= \tfrac{1}{2}a_{11} + \tfrac{1}{2}a_{11} e_1 + \tfrac{1}{2}a_{21} e_2 - \tfrac{1}{2}a_{21} e_1 e_2 + \\ &\quad \tfrac{1}{2}a_{12} e_2 + \tfrac{1}{2}a_{12} e_1 e_2 + \tfrac{1}{2}a_{22} - \tfrac{1}{2}a_{22} e_1 \end{aligned}$$

Equating coefficients it is clear that the following equations apply.

$$\begin{aligned} a_0 &= \tfrac{1}{2}a_{11} + \tfrac{1}{2}a_{22} \\ a_1 &= \tfrac{1}{2}a_{11} - \tfrac{1}{2}a_{22} \\ a_2 &= \tfrac{1}{2}a_{12} + \tfrac{1}{2}a_{21} \\ a_3 &= \tfrac{1}{2}a_{12} - \tfrac{1}{2}a_{21} \end{aligned}$$

The reverse equations can be recovered by multiplying the two forms of c by different combinations of the functions $f_1$, $f_2$ and $f_3$. The equation

$$\begin{aligned} f_1 c f_1 &= f_1(a_{11} f_1 + a_{21} f_2 + a_{12} f_3 + a_{22} f_4) f_1 \\ &= f_1(a_0 + a_1 e_1 + a_2 e_2 + a_3 e_1 e_2) f_1 \end{aligned}$$

reduces to the equation

$$a_{11}f = (a_0 + a_1)f$$

and similar equations can be deduced from other combinations of the functions as follows.

$$\begin{aligned} f_1 c f_2 : a_{12}f &= (a_2 + a_3)f \\ f_2 c f_1 : a_{21}f &= (a_2 - a_3)f \\ f_3 c f_2 : a_{22}f &= (a_0 - a_1)f \end{aligned}$$

If a matrix is defined as

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

so that

$$Af = \begin{pmatrix} a_{11}f & a_{12}f \\ a_{21}f & a_{22}f \end{pmatrix} = \begin{pmatrix} a_0 + a_1 & a_2 + a_3 \\ a_2 - a_3 & a_0 - a_1 \end{pmatrix} f$$

then the expression

$$\begin{pmatrix} 1 & e_2 \end{pmatrix} \begin{pmatrix} a_{11}f & a_{12}f \\ a_{21}f & a_{22}f \end{pmatrix} \begin{pmatrix} 1 \\ e_2 \end{pmatrix} = a_{11}f_1 + a_{21}f_2 + a_{12}f_3 + a_{22}f_4 = c$$

generates the general Clifford object c. All that remains to form the basis matrices is to make c each basis member in turn, and named as shown.

$$\begin{aligned} c = 1 : \quad Af &= \begin{pmatrix} f & 0 \\ 0 & f \end{pmatrix} &&= E_0 f \\ c = e_1 \quad Af &= \begin{pmatrix} f & 0 \\ 0 & -f \end{pmatrix} &&= E_1 f \\ c = e_2 \quad Af &= \begin{pmatrix} 0 & f \\ f & 0 \end{pmatrix} &&= E_2 f \\ c = e_1 e_2 \quad Af &= \begin{pmatrix} 0 & f \\ -f & 0 \end{pmatrix} &&= E_{12} f \end{aligned}$$

These are the usual basis matrices for Clifford (2) except that they are multiplied by the idempotent.

This approach provides an explanation for the basis matrices in terms only of the Clifford Algebra itself. They are the matrix representation of the basis objects of the algebra in terms of an idempotent and an associated vector of basis functions. This has been shown for Clifford (2) and it can be extended to other algebras once the idempotent and the vector of basis functions have been identified. This has been done in many cases by Ablamowicz. This will now be developed to show how the inverse of a Clifford number can be obtained from the matrix representation.

## 5.3 Calculation of the inverse of a Clifford number

The matrix basis demonstrated above can be used to calculate the inverse of a Clifford number. In simple cases this can be used to obtain an algebraic formulation. For other cases the algebra is too complex to be clear, but the method can still be used to obtain the numerical value of the inverse. To apply the method it is necessary to know a basis matrix representation of the algebra being used.

The idea of the method is that the matrix representation will have a characteristic polynomial obeyed by the eigenvalues of the matrix and also by the matrix itself. There may also be a minimal polynomial which is a factor of the characteristic polynomial, which will have also be satisfied by the matrix. It is clear from the proceding section that if $A$ is a matrix representation of $c$ in a Clifford Algebra then if some function $f(A) = 0$ then the corresponding Clifford function $f(c) = 0$ must also be zero. In particular if $f(A) = 0$ is the characteristic or minimal polynomial of $A$, then $f(c) = 0$ implies that $c$ also satisfies the same polynomial. Then if the inverse of the Clifford number, $c^{-1}$ is to be found, then

$$c^{-1} f(c) = 0$$

provides a relationship for $c^{-1}$ in terms of multiples a small number of low powers of $c$, with the maximum power one less than the order of the polynomial. The method suceeds unless the constant term in the polynomial is zero, which means that the inverse does not exist. For cases where the basis matrices are of order two, the inverse will be shown to be a linear function of $c$.

The method can be summed up as follows.

1. Find the matrix basis of the Clifford algebra.
2. Find the matrix representation of the Clifford number whose inverse is required.
3. Compute the characteristic or minimal polynomial.
4. Check for the existence of the inverse.
5. Compute the inverse using the coefficients from the polynomial.

Step 1 need only be done once for any Clifford algebra, and this can be done using the method in the previous section, where needed.

Step 2 is trivially a matter of accumulation of the correct multiples of the matrices.

Step 3 may involve the use of a computer algebra system to find the coefficients of the polynomial, if the matrix size is at all large.

Steps 4 and 5 are then easy once the coefficients are known.

The method will now be demonstrated using some examples.

### 5.3.1 Example 1: Clifford (2)

In this case the matrix basis for a member of the Clifford algebra

$$c = a_0 + a_1 e_1 + a_2 e_2 + a_3 e_1 e_2$$

was developed in the previous section as

$$A = \begin{pmatrix} a_0 + a_1 & a_2 + a_3 \\ a_2 - a_3 & a_0 - a_1 \end{pmatrix}$$

This matrix has the characteristic polynomial

$$X^2 - 2Xa_0 + a_0^2 - a_1^2 - a_2^2 + a_3^2 = 0$$

and therefore

$$X^{-1}(X^2 - 2Xa_0 + a_0^2 - a_1^2 - a_2^2 + a_3^2) = 0$$

and

$$X^{-1} = (2a_0 - X)/(a_0^2 - a_1^2 - a_2^2 + a_3^2) = 0$$

which provides a general solution to the inverse in this algebra.

$$c^{-1} = (2a_0 - c)/(a_0^2 - a_1^2 - a_2^2 + a_3^2) = 0$$

### 5.3.2 Example 2: Clifford (3)

A set of basis matrices for Clifford (3) as given by Abalmowicz and deduced are

$$\begin{array}{ll} E_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & E_1 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \\ E_2 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} & E_3 = \begin{pmatrix} 0 & -j \\ j & 0 \end{pmatrix} \\ E_1E_2 = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} & E_1E_3 = \begin{pmatrix} 0 & -j \\ -j & 0 \end{pmatrix} \\ E_2E_3 = \begin{pmatrix} j & 0 \\ 0 & -j \end{pmatrix} & E_1E_2E_3 = \begin{pmatrix} j & 0 \\ 0 & j \end{pmatrix} \end{array}$$

for the idempotent

$$f = \frac{(1 + e_1)}{2}, \text{ where } j^2 = -1.$$

The general member of the algebra

$$c_3 = a_0 + a_1e_1 + a_2e_2 + a_3e_3 + a_{12}e_1e_2 + a_{13}e_1e_3 + a_{23}e_2e_3 + a_{123}e_1e_2e_3$$

has the matrix representation

$$\begin{aligned} A_3 &= a_0E_0 + a_1E_1 + a_2E_2 + a_3E_3 + a_{12}E_1E_2 \\ &\quad + a_{13}E_1E_3 + a_{23}E_2E_3 + a_{123}E_1E_2E_3 \\ &= \begin{pmatrix} a_0 + a_1 + ja_{23} + ja_{123} & a_2 - ja_3 + a_{12} - ja_{13} \\ a_2 + ja_3 - a_{12} - ja_{13} & a_0 - a_1 - ja_{23} + ja_{123} \end{pmatrix} \end{aligned}$$

This has the characteristic polynomial

$$\begin{aligned} & a_0^2 - a_1^2 - a_2^2 - a_3^2 + a_{12}^2 + a_{13}^2 + a_{23}^2 - a_{123}^2 \\ + \quad & 2j(a_0 a_{123} - a_1 a_{23} - a_{12} a_3 + a_{13} a_2) \\ - \quad & 2(a_0 + j a_{123})X + X^2 = 0 \end{aligned}$$

and the expression for the inverse is

$$\begin{aligned} X^{-1} \quad = \quad & (2a_0 + 2j a_{123} - X)/ \\ & (a_0^2 - a_1^2 - a_2^2 - a_3^2 + a_{12}^2 + a_{13}^2 + a_{23}^2 - a_{123}^2 \\ & +2j(a_0 a_{123} - a_1 a_{23} - a_{12} a_3 + a_{13} a_2)) \end{aligned}$$

Complex terms arise in two cases,

$$a_{123} \neq 0$$

and

$$(a_0 a_{123} - a_1 a_{23} - a_{12} a_3 + a_{13} a_2) \neq 0$$

Two simple cases have real minumum polynomials:
Zero and first grade terms only:

$$\begin{aligned} A_1 \quad = \quad & a_0 E_0 + a_1 E_1 + a_2 E_2 + a_3 E_3 \\ = \quad & \begin{pmatrix} a_0 + a_1 & a_2 - j a_3 \\ a_2 + j a_3 & a_0 - a_1 \end{pmatrix} \end{aligned}$$

which has the minimum polynomial

$$a_0^2 - a_1^2 - a_2^2 - a_3^2 - 2a_0 X + X^2 = 0$$

which gives

$$X^{-1} = (2a_0 - X)/(a_0^2 - a_1^2 - a_2^2 - a_3^2)$$

Zero and second grade terms only (ie. the even subspace).

$$\begin{aligned} A_2 \quad = \quad & a_0 E_0 + a_{12} E_1 E_2 + a_{13} E_1 E_3 + a_{23} E_2 E_3 \\ & \begin{pmatrix} a_0 + j a_{23} & a_{12} - j a_{13} \\ -a_{12} - j a_{13} & a_0 - j a_{23} \end{pmatrix} \end{aligned}$$

which has minimum polynomial

$$a_0^2 + a_{23}^2 + a_{12}^2 + a_{13}^2 - 2a_0 X + X^2 = 0$$

giving

$$X^{-1} = (2a_0 - X)/(a_0^2 + a_{23}^2 + a_{12}^2 + a_{13}^2)$$

This provides a general solution for the inverse together with two simple cases of wide usefulness.

### 5.3.3 Example 3: Clifford (2,2)

The following basis matrices are given by Ablamowicz [Abla98]

$$E_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad E_2 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix}$$
$$E_3 = \begin{pmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad E_4 = \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix}$$

for the idempotent

$$f = \frac{(1 + e_1e_3)(1 + e_1e_3)}{4}.$$

Note that this implies that the order of the basis members is such that $e_1$ and $e_2$ have square $+1$ and $e_3$ and $e_4$ have square $-1$. Other orderings are used by other authors. The remaining basis matrices can be deduced to be as follows.

Second Grade members

$$E_1E_2 = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \quad E_1E_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$
$$E_1E_4 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \quad E_2E_3 = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix}$$
$$E_2E_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \quad E_3E_4 = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Third grade members

$$E_1E_2E_3 = \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix} \quad E_1E_2E_4 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

$$E_1E_3E_4 = \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \quad E_2E_3E_4 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Fourth grade member

$$E_1E_2E_3E_4 = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

Zero grade member (identity)

$$E_0 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The general member of the Clifford (2,2) algebra can be written as follows.

$$\begin{aligned} c_{22} \quad = \quad & a_0 + a_1e_1 + a_2e_2 + a_3e_3 + a_4e_4 + \\ & a_{12}e_1e_2 + a_{13}e_1e_3 + a_{14}e_1e_4 + a_{23}e_2e_3 + a_{24}e_2e_4 + a_{34}e_3e_4 \\ & + a_{123}e_1e_2e_3 + a_{124}e_1e_2e_4 + a_{134}e_1e_3e_4 + a_{234}e_2e_3e_4 + a_{1234}e_1e_2e_3e_4 \end{aligned}$$

This has the following matrix representation.

$$\begin{pmatrix} a_0 + a_{13} + & a_1 - a_3 + & a_2 - a_4 - & -a_{12} + a_{14} - \\ a_{24} - a_{1234} & a_{124} + a_{234} & a_{123} - a_{134} & a_{23} - a_{34} \\ \\ a_1 + a_3 + & a_0 - a_{13} + & a_{12} - a_{14} - & -a_2 + a_4 - \\ a_{124} - a_{234} & a_{24} + a_{1234} & a_{23} - a_{34} & a_{123} - a_{134} \\ \\ a_2 + a_4 - & -a_{12} - a_{14} - & a_0 + a_{13} - & a_1 - a_3 - \\ a_{123} + a_{134} & a_{23} + a_{34} & a_{24} + a_{1234} & a_{124} - a_{234} \\ \\ a_{12} + a_{14} - & -a_2 - a_4 - & a_1 + a_3 - & a_0 - a_{13} - \\ a_{23} + a_{34} & a_{123} + a_{134} & a_{124} + a_{234} & a_{24} - a_{1234} \end{pmatrix}$$

In this case it is possible to generate the characteristic equation using computer algebra. However, it is too complex to be of practical use. Instead here are numerical examples of the use of the method to calculate the inverse. For the case where

$$n1 = 1 + e_1 + e_2 + e_3 + e_4$$

then the matrix representation is

$$N_1 = E_0 + E_1 + E_2 + E_3 + E_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 2 & 0 & 1 & 0 \\ 0 & -2 & 2 & 1 \end{pmatrix}$$

This has the minimum polynomial

$$X^2 - 2X + 1 = 0$$

so that

$$X^{-1} = 2 - X$$

and

$$n_1^{-1} = 2 - n_1 = 1 - e_1 - e_2 - e_3 - e_4$$

For

$$n_2 = 1 + e_1 + e_2 + e_3 + e_4 + e_1 e_2$$

the matrix representation is

$$N_2 = I + E_1 + E_2 + E_3 + E_4 + E_1 E_2 = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 2 & 1 & 1 & 0 \\ 2 & -1 & 1 & 0 \\ 1 & -2 & 2 & 1 \end{pmatrix}$$

This has the minimum polynomial

$$X^4 - 4X^3 + 8X^2 - 8X - 4 = 0$$

so that

$$X^{-1} = \frac{X^3 - 4X^2 + 8X - 8}{4}$$

and

$$n_2^{-1} = \frac{n_2^3 - 4n_2^2 + 8n_2 - 8}{4}$$

This expression can be evaluated easily using a computer algebra system for Clifford algebra such as described in Fletcher [Flet01]. The result is

$$\begin{aligned} n_2^{-1} \quad = \quad & -0.5 + 0.5e_1 + 0.5e_2 - 0.5e_1e_2 - 0.5e_1e_3 \\ & -0.5e_1e_4 + 0.5e_2e_3 + 0.5e_2e_4 - 0.5e_1e_2e_3 - 0.5e_1e_2e_4 \end{aligned}$$

Note that in some cases the inverse is linear in the original Clifford number, and in others it is nonlinear.

### 5.3.4 Conclusion

The paper has demonstrated a method for the calculation of inverses of Clifford numbers by means of the matrix representation of the corresponding Clifford algebra. The method depends upon the calculation of the basis matrices for the algebra. This can be done from an idempotent for the algebra if the matrices are not already available. The method provides an easy check on the existence of the inverse. For simple systems a general algebraic solution can be found and for more complex systems the algebra of the inverse can be generated and evaluated numerically for a particular example, given a system of computer algebra for Clifford algebra.

# Chapter 6

# Package for Algebraic Function Fields

PAFF is a Package for Algebraic Function Fields in one variable by Gaétan Haché

PAFF is a package written in Axiom and one of its many purpose is to construct geometric Goppa codes (also called algebraic geometric codes or AG-codes). This package was written as part of Gaétan's doctorate thesis on "Effective construction of geometric codes": this thesis was done at Inria in Rocquencourt at project CODES and under the direction of Dominique LeBrigand at Universit Pierre et Marie Curie (Paris 6). Here is a résumé of the thesis.

It is well known that the most difficult part in constructing AG-code is the computation of a basis of the vector space "L(D)" where D is a divisor of the function field of an irreducible curve. To compute such a basis, PAFF used the Brill-Noether algorithm which was generalized to any plane curve by D. LeBrigand and J.J. Risler [LeBr88]. In [Hach96] you will find more details about the algorithmic aspect of the Brill-Noether algorithm. Also, if you prefer, as I do, a strictly algebraic approach, see [Hach95]. This is the approach I used in my thesis ([Hach96]) and of course this is where you will find complete details about the implementation of the algorithm. The algebraic approach use the theory of algebraic function field in one variable : you will find in [Stic93] a very good introduction to this theory and AG-codes.

It is important to notice that PAFF can be used for most computation related to the function field of an irreducible plane curve. For example, you can compute the genus, find all places above all the singular points, compute the adjunction divisor and of course compute a basis of the vector space L(D) for any divisor D of the function field of the curve.

There is also the package PAFFFF which is especially designed to be used over finite fields. This package is essentially the same as PAFF, except that the computation are done over "dynamic extensions" of the ground field. For this, I used a simplify version of the notion of dynamic algebraic closure as proposed by D. Duval [Duva95].

Example 1

This example compute the genus of the projective plane curve defined by:

```
 5    2 3      4
X  + Y Z  + Y Z  = 0
```

over the field GF(2).

First we define the field GF(2).

```
K:=PF 2
R:=DMP([X,Y,Z],K)
P:=PAFF(K,[X,Y,Z],BLQT)
```

We defined the polynomial of the curve.

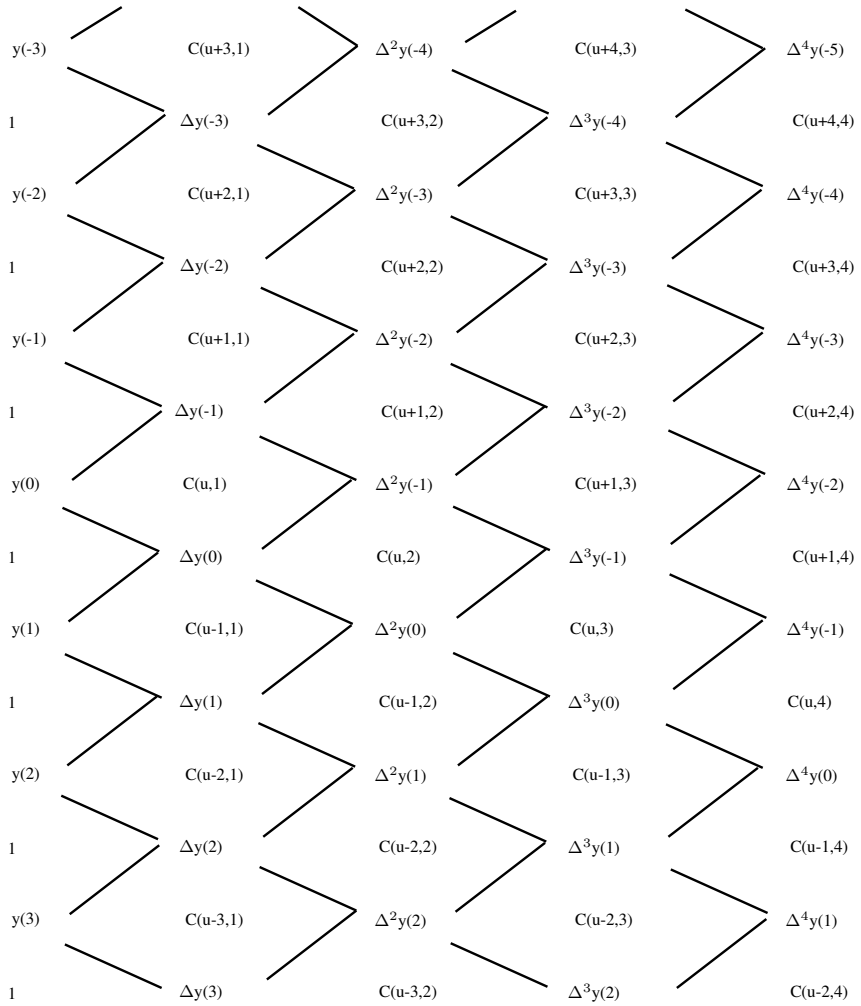
```
C:R:=X**5 + Y**2*Z**3+Y*Z**4
```

We give it to the package PAFF(K,[X,Y,Z]) which was assigned to the variable $P$.

```
setCurve(C)$P
```

# Chapter 7

# Interpolation Formulas




The lozenge diagram is a device for showing that a large number of formulas which appear

to be different are really all the same. The notation for the binomial coefficients

$$C(u+k,n) = \frac{(u+k)(u+k-1)(u+k-2)\cdots(u+k-n+1)}{n!}$$

There are $n$ factors in the numerator and $n$ in the denominator. Viewed as a function of $u$, $C(u+k,n)$ is a polynomial of degree $n$.

The figure above, Hamming [Hamm62] calls a lozenge diagram. A line starting at a point on the left edge and following some path across the page defines an interpolation formula if the following rules are used.

**1a** For a left-to-right step, *add*

**1b** For a right-to-left, *subtract*

**2a** If the *slope* of the step is *positive*, use the product of the difference crossed times the factor immediately *below*.

**2b** If the *slope* of the step is *negative*, use the product of the difference crossed times the factor immediately *above*

**3a** If the step is *horizontal* and passes through a *difference*, use the product of the difference times the *average* of the factors *above* and *below*.

**3b** If the step is *horizontal* and passes through a *factor*, use the product of the factor times the *average* of the differences *above* and *below*.

As an example of rules **1a** and **2a**, consider starting at $y(0)$ and going down to the right. We get, term by term,

$$y(u) = y(0) + C(u,1)\Delta y(0) + C(u,2)\Delta^2 y(0) + C(u,3)\Delta^3 y(0) + \cdots$$

$$= y(0) + u\Delta y(0) + \frac{u(u-1)}{2}\Delta^2 y(0) + \frac{u(u-1)(y-2)}{3!}\Delta^3 y(0) + \cdots$$

which is Newton's formula.

Had we gone up and to the right, we would have used **1a** and **2a** to get Newton's backward formula:

$$y(u) = y(0) + C(u,1)\Delta y(-1) + C(u+1,2)\Delta^2 y(-2) + C(u+2,3)\Delta^3 y(-3) + \cdots$$

$$= y(0) + u\Delta y(-1) + \frac{(u+1)u}{2}\Delta^2 y(-2) + \frac{(u+2)(u+1)u}{3!}\Delta^3 y(-3) + \cdots$$

To get Stirling's formula, we start at $y(0)$ and go horizontally to the right, using rules **3a** and **3b**:

$$y(u) = y(0) + u\frac{\Delta y_0 + \Delta y_{-1}}{2} + \frac{C(u+1,2) + C(u,2)}{2}\Delta^2 y_{-1} + C(u+1,3)\frac{\Delta^3 y_{-2} + \Delta^3 y_{-1}}{2} + \cdots$$

$$= y_0 + u\frac{\Delta y_0 + \Delta y_{-1}}{2} + \frac{u^2}{2}\Delta^2 y_{-1} + \frac{u(u^2-1)}{3!}\frac{\Delta^3 y_{-2} + \Delta^3 y_{-1}}{2} + \cdots$$

If we start midway between $y(0)$ and $y(1)$, we get Bessel's formula:

$$y(u) = 1\frac{y_0 + y_1}{2} + \frac{C(u,1) + C(u-1,1)}{2}\Delta y_0 + C(u,2)\frac{\Delta^2 y_{-1} + \Delta^2 y_0}{2} + \cdots$$

$$= \frac{y_0 + y_1}{2} + (u - \frac{1}{2})\Delta y_0 + \frac{u(u-1)}{2}\frac{\Delta^2 y_{-1} + \Delta^2 y_0}{2} + \cdots$$

If we zigzag properly, we can get Gauss' formula for interpolation:

$$y(u) = y_0 + u\Delta y_0 + \frac{u(u-1)}{2}\Delta^2 y(-1) + \frac{u(u^2-1)}{3!}\Delta^3 y(-1) + \cdots$$

# Chapter 8

# Type Systems for Computer Algebra by Andreas Weber

This chapter is based on a PhD thesis by Andreas Weber[Webe93b]. Changes have been made to integrate it.

We study type systems for computer algebra systems, which frequently correspond to the "pragmatically developed" typing constructs used in Axiom.

A central concept is that of *type classes* which correspond to Axiom categories. We will show that types can be syntactically described as terms of a regular order-sorted signature if no type parameters are allowed. Using results obtained for the functional programming language Haskell we will show that the problem of *type inference* is decidable. This result still holds if higher-order functions are present and *parametric polymorphism* is used. These additional typing constructs are useful for further extensions of existing computer algebra systems: These typing concepts can be used to implement category theoretic constructs and there are many well known constructive interactions between category theory and algebra.

On the one hand we will show that there are well known techniques to specify many important type classes algebraically, and we will also show that a formal and algorithmically Feasible treatment of the interactions of algebraically specified data types and type classes is possible. On the other hand we will prove that there are quite elementary examples arising in computer algebra which need very "strong" formalisms to be specified and are thus hard to handle algorithmically.

We will show that it is necessary to distinguish between types and elements as parameters of parameterized type classes. The type inference problem for the former remains decidable whereas for the latter it becomes undecidable. We will also show that such a distinction can be made quite naturally.

Type classes are second-order types. Although we will show that there are constructions used in mathematics which imply that type classes have to become first-order types in order to model the examples naturally, we will also argue that this does not seem to be the case in areas currently accessible for an algebra system. We will only sketch some systems that have been developed during the last years in which the concept of type classes as first-order types can be expressed. For some of these systems the type inference problem was proven to be undecidable.

Another fundamental concept for a type system of a computer algebra system — at least for the purpose of a user interface — are *coercions*. We will show that there are cases which can be modeled by coercions but not by an "inheritance mechanism", i. e. the concept of coercions is not only orthogonal to the one of type classes but also to more general formalisms as are used in object-oriented languages. We will define certain classes of coercions and impose conditions on important classes of coercions which will imply that the meaning of an expression is independent of the particular coercions that are used in order to type it.

We will also impose some conditions on the interaction between polymorphic operations defined in type classes and coercions that will yield a unique meaning of an expression independent of the type which is assigned to it — if coercions are present there will very frequently be several possibilities to assign types to expressions.

Often it is not only possible to coerce one type into another but it will be the case that two types are actually *isomorphic*. We will show that isomorphic types have properties that cannot be deduced from the properties of coercions and will shortly discuss other possibilities to model type isomorphisms. There are natural examples of type isomorphisms occurring in the area of computer algebra that have a "problematic" behavior. So we will prove for a certain example that the type isomorphisms cannot be captured by a finite set of coercions by proving that the naturally associated equational theory is not finitely axiomatizable.

Up to now few results are known that would give a clear dividing line between classes of coercions which have a decidable type inference problem and classes for which type inference becomes undecidable. We will give a type inference algorithm for some important classes of coercions.

Other typing constructs which are again quite orthogonal to the previous ones are those of *partial functions* and of *types depending on elements*. We will link the treatment of *partial functions* in Axiom to the one used in order-sorted algebras and will show some problems which arise if a seemingly more expressive solution were used. There are important cases in which *types depending on elements* arise naturally. We will show that not only type inference but even type checking is undecidable for relevant cases occurring in computer algebra.

Types have played an extremely important role in the development and study of programming languages. They have become so prevalent that type theory is now recognized as an area of its own within computer science. The benefits which can be derived from the presence of types in a language are manifold. Through type checking many errors can be caught before a program is ever run, thus leading to more reliable programs. Types form also an expressive basis for module systems, since they prescribe a machine-verifiable interface for the code encapsulated within a module. Furthermore, they may be used to improve performance of code generated by a compiler.

However, most computer algebra systems are based on untyped languages. Nevertheless, at least in the description and specification of many algorithms a terminology is used which can be seen as attributing "types" to the computational objects. In Maple V [Char91] and in Mathematica [Wolf91], which are both based on untyped languages, it is even possible to attach "tags" to data structures which describe types corresponding to the mathematical structures the data are supposed to represent.

In the area of computer algebra, the problem of finding appropriate type systems which are supported by the language is that on the one hand, the type system has to consider the requirements of a computer system and on the other, it should allow for the mathematical structures a system is dealing with to have corresponding types.

The development of Axiom [Jenk84b], [Suto87], [Jenk92] is certainly a break-through since

the language itself is typed with types corresponding to the mathematical structures the system deals with.

However, the typing constructs used in Axiom have been "pragmatically developed." Some are not even formally defined and only very few studies on formal properties of such a system have been undertaken. Even if other approaches to a type system in this area are considered — such as the "object-oriented" one used for VIEWS [Abda86] — we have found relatively few formal studies of type systems suited for the purpose of computer algebra systems in the literature, although a formal treatment of some typing constructs occurring in computer algebra was already given almost twenty years ago in [Loos74].

So the situation is different from the one in other areas of computer science in which untyped languages are prevalent. For instance, most logic programming languages are untyped. This is a consequence of the fact that logic programming has its roots in first-order logic, which is essentially untyped. Nevertheless, the progress of type theory in the last decade has allowed the development of several type systems for logic programming languages. Moreover, the formal properties of these type systems have been studied extensively (see e. g. [Smol89a], [Frue91], [Kife91], and the articles in the collection [Pfen92], in which also a comprehensive bibliography on the topic is given).

We will not design a typed computer algebra language in this thesis in which the mathematical structures a program deals with have a correspondence in the type system. It does not seem possible to design and implement a language of similar power as Axiom within a PhD-project. There are several proposals of languages for computer algebra systems[1] which are designed and partly implemented as part of a PhD-project that incorporate some typing concepts, but which can be seen — more or less — as subsets of the typing constructs of Axiom.

Instead we will treat typing constructs which are similar in power to the ones of Axiom. We will define type systems of various strength and will investigate their properties. Discussing a variety of examples we will show their relevance for a computer algebra system. We will also discuss some examples which are not implemented in a system as yet in order to give some estimates about the extendability of a system based on such typing principles. This is one of the shortcomings of many other investigations in which very often only examples that can be modeled are discussed. We hope that our discussion of a variety of examples will help to obtain characterization theorems of mathematical structures which can be modeled by certain typing constructs. This would be the best solution. However, it seems to be a large-scale task to obtain such characterization theorems in many cases. A problem in this connection is certainly that one has to define precisely a class of mathematical structures a program is dealing with at all. Current computer algebra programs sometimes deal with objects of universal algebra, sometimes with those of higher-order universal algebra, sometimes with those of first-order model theory, or sometimes with those of category theory, to mention only some possibilities.

We will prove several properties of such type systems. A very important feature is the possibility of *type inference*. Given an expression the system should be able to infer a correct type for it whenever possible and reject it otherwise. Since the interpretation of an expression written in the standard mathematical notation requires a kind of type inference very frequently the possibility of type inference improves considerably the usefulness of a system for a user. Thus we will investigate the problems connected with type inference

[1] The author knows of Foderaro's NEWSPEAK [Fode83], Coolsaet's MIKE [Cool92], and Dalmas' XFun [Dalm92].

extensively and will also give some results on the computational complexity of various type inference problems. Another important problem we will investigate in various, precisely defined ways is a possible ambiguity of a type system.

Some of the results we give are contained in some form in the literature, especially in papers on type systems for functional languages. Nevertheless, it seems to have escaped prior notice that these results are applicable to the typing problems arising in computer algebra.

On the one hand it is useful to have a system which can handle as many mathematical structures as possible. For many mathematicians a computer algebra system would be a very valuable tool if it allowed some computations in rather complicated mathematical structures. Since many of those computations would be fairly basic it would suffice for these users to have a system in which they could model those structures easily, even if that modeling was not very efficient. Among the existing systems Axiom is one of the few which gives the possibility for such work.[2] So it seems to be necessary to have a safe foundation for the constructs found in such a universal system as Axiom.

On the other hand many computations that have to be performed reach the limits of existing computing power. So the algorithms should be as efficient as possible in order to be useful. Since it seems to be impossible to have a general system that is always as efficient as a more special one — and this thesis will contain some results which can be viewed as a proof of this claim — we will not only develop a framework for a general computer algebra system and discuss its properties but will also discuss the properties of some subsystems. The author hopes that some of these results will be useful for the design of symbolic manipulation systems or the design of user interfaces for such systems.

The organization of the thesis will be as follows.

In Sec. 8.1 we will collect some definitions and facts which will be needed later. Most of the material in this chapter can be found scattered in the literature. Moreover, we will fix the notation and will give some discussion on the terminology used in this thesis as compared to the one found in the literature.

A central concept is that of *type classes* which correspond to Axiom categories and will be the subject of Sec. 8.2.[3] We will show that types can be syntactically described as terms of a regular order-sorted signature if no type parameters are allowed. Using results obtained for the functional programming language Haskell we will show that the problem of *type inference* is decidable. This result still holds if higher-order functions are present and *parametric polymorphism* is used. These additional typing constructs are useful for further extensions of existing computer algebra systems: These typing concepts can be used to implement category theoretic constructs and there are many well known constructive interactions between category theory and algebra.

On the one hand we will show that there are well known techniques to specify many important type classes algebraically, and we will also show that a formal treatment of the interactions of algebraically specified data types and type classes is possible. On the other hand we will prove that there are quite elementary examples arising in computer algebra which need very "strong" formalisms to be specified.

[2] The new version of Cayley [Butl90] allows similar possibilities but fewer structures have been implemented as yet.

[3] They are similar to the *varieties* of Cayley, if a Cayley *class* is interpreted as a type, which can be done using the concept of *types depending on elements* (see below). They are also similar to *container classes* used in object-oriented programming. However, we will not give a systematic treatment of constructs of object-oriented programming in this thesis.

We will show that it is necessary to distinguish between types and elements as parameters of parameterized type classes. The type inference problem for the former remains decidable whereas for the latter it becomes undecidable. We will also show that such a distinction can be made quite naturally.

Type classes are second-order types. Although we will show that there are constructions used in mathematics which imply that type classes have to become first-order types in order to model the examples naturally, we will also argue that this does not seem to be the case in areas currently accessible for an algebra system. We will only sketch some systems that have been developed during the last years in which the concept of type classes as first-order types can be expressed. For some of these systems the type inference problem was proven to be undecidable, thus showing one of the drawbacks of stronger formalisms.

In Sec. 8.3 we will treat the concept of *coercions* which is another fundamental concept for a type system of a computer algebra system, at least for the purpose of a user interface. We will show that there are cases which can be modeled by coercions but not by an "inheritance mechanism", i. e. the concept of coercions is not only orthogonal to the one of type classes but also to formalisms extending type classes. We will define certain classes of coercions and impose conditions on important classes of coercions which will imply that the meaning of an expression is independent of the particular coercions that are used in order to type it. These results will also appear in [Webe95].

We will also impose some conditions on the interaction between polymorphic operations defined in type classes and coercions that will yield a unique meaning of an expression independent of the type which is assigned to it — if coercions are present there will very frequently be several possibilities to assign types to expressions.

Often it is not only possible to coerce one type into another but it will be the case that two types are actually *isomorphic*. We will show that isomorphic types have properties that cannot be deduced from the properties of coercions and will shortly discuss other possibilities to model type isomorphisms.

Unfortunately, there are natural examples of type isomorphisms occurring in the area of computer algebra that have a "problematic" behavior. For a major example of types having type isomorphisms that cannot be captured by a finite set of coercions, we will provide a proof that no such finite set can be given by proving that the naturally associated equational theory is not finitely axiomatizable. This example and the given proof are published by the author in [Webe05].

We will give a semi-decision procedure for type inference for a system having type classes and coercions and a decision procedure for a subsystem which covers many important cases occurring in computer algebra. Up to now few results are known that would give a clear dividing line between classes of coercions which have a decidable type inference problem and classes for which type inference becomes undecidable. However, even in decidable cases the type inference problem in the presence of coercions is a hard problem. Even in cases in which the possible coercions are rather restricted the type inference problem was proven to be NP-hard for functional languages.

Two typing constructs which are again quite orthogonal to the previous ones are treated in Sec. 8.4. We will link the treatment of *partial functions* in Axiom to the one used in order-sorted algebras and will show some problems which arise if a seemingly more expressive solution were used. Nevertheless, some information is lost by the used solution and we sketch a proposal how the lost information could be regained in certain cases.

There are important cases in which *types depending on elements* arise naturally. Unfortu-

nately, not only type inference but even type checking are undecidable for relevant cases occurring in computer algebra, i. e. static type checking is not possible. On the one hand we will show that already types which have to be given to the objects in standard algorithms of almost any general purpose computer algebra program will prohibit static type checking. On the other hand it might be possible to restrict the types depending on elements available to a user of a high-level user interface to classes which have decidable type checking or even type inference problems. We will show that several formalisms have been developed during the last years which might be relevant in this respect.

## 8.1 Prelude

We will recall some definitions and facts which will be needed later. All of this material can be found scattered in the literature. Moreover, we will fix the notation and will give some discussions of the terminology used in this thesis in comparison to the one found in the literature.

### 8.1.1 Terminology

#### Abstract Data Types

The term *data type* has many informal usages in programming and programming methodology. For instance, Gries lists seven interpretations in [Grie78].

In this thesis we will deal with different meanings of the term *abstract data type* (ADT). On the one hand there is the meaning used in the context of algebraic specifications as it is used e. g. in the survey of Wirsing [Wirs91]. In this context an abstract datatype given by a specification is a class of certain many-sorted (or order-sorted) algebras which "satisfy" the specification.

On the other hand there is the usage of this term for data types whose representation is hidden. For instance, in the report on the language Haskell [Huda92] the authors state "the characteristic feature of an ADT is that the *representation type is hidden*; all operations on the ADT are done at an abstract level which does not depend on the representation". The explanation given in the glossary of the book on Axiom [Jenk92] is quite similar:

> **abstract datatype**
> a programming language principle used in Axiom where a datatype definition has defined in two parts: (1) a *public* part describing a set of *exports*, principally operations that apply to objects of that type, and (2) a *private* part describing the implementation of the datatype usually in terms of a *representation* for objects of the type. Programs that create and otherwise manipulate objects of the type may only do so through its exports. The representation and other implementation information is specifically hidden.

Usually the purpose of abstract data types in the sense of algebraic specifications is for the specification of abstract data types in the sense of the quotations given above. However, as we will show in this thesis, the abstract data types in the former sense can also be used for the specification of other classes of computational objects than abstract data types in the latter sense.

### Polymorphism

Although the term *polymorphic function* is used in the literature, there are usually no definitions given.

In the glossary of [Jenk92] only examples of polymorphic functions are given but no definition. Also in the book by Aho, et al. [Ahox86, p. 364], the term is explained by giving examples of polymorphic functions.

In the recent survey of Mitchell [Mitc91a] the author states explicitly that he does not want to give a definition of *polymorphism*, but that he will only give definitions of some "polymorphic lambda-calculi".

There is a distinction between *parametric polymorphism* and *ad hoc polymorphism* which seems to go back to Strachey [Stra00] (cited after [Gogu89]):

> In *ad hoc* polymorphism there is no simple systematic way of determining the type of the result from the type of the arguments. There may be several rules of limited extent which reduce the number of cases, but these are themselves *ad hoc* both in scope and in content. All the ordinary arithmetic operations and functions come into this category. It seems, moreover, that the automatic insertion of transfer functions by the compiling system is limited to this class.
>
> Parametric polymorphism is more regular and may be illustrated by an example. Suppose f is a function whose arguments is of type $\alpha$ and whose result is of type $\beta$ (so that the type of f might be written $\alpha \longrightarrow \beta$, and that L is a list whose elements are all of type $\alpha$ (so that the type of L is $\alpha$**list**). We can imagine a function, say Map, which applies f in turn to each member of L and makes a list of the results. Thus Map[f,L] will produce a $\beta$**list**. We would like Map to work on all types of list provided f was a suitable function, so that Map would have to be polymorphic. However its polymorphism is of a particularly simple parametric type which could be written $(\alpha \longrightarrow \beta, \alpha\textbf{list}) \longrightarrow \beta\textbf{list}$, where $\alpha$ and $\beta$ stand for any types.

A widely accepted approach to parametric polymorphism is the Hindley-Milner type system [Hind69], [Miln78], [Dama82], which is used in Standard ML [Miln90], [Miln91], Miranda [Turn85], [Turn86] and other languages.

We will use the term parametric polymorphism in this sense.

There is no widely accepted approach to ad-hoc polymorphism. In its general form, we will use the word ad-hoc polymorphism and overloading quite synonymously indicating that no restriction is imposed on the possibility to overload an operator symbol.

However, there is a third form of polymorphism which will play a central role in this thesis and for which an appropriate name is missing. It is the polymorphism which occurs when *categories* in the Axiom-terminology resp. *type classes* in the Haskell-terminology are used. In [Wadl88] the nice negative formulation "How to make *ad-hoc* polymorphism less *ad-hoc*" is used but no proposal for a positive name is given. When necessary we will call the polymorphism encountered by type classes simply *type-class polymorphism*.[4]

Sometimes a distinction is made between *polymorphic functions* and *generic function calls*. The intended meaning — e. g. in [Fode83] — is that *polymorphic* refers to functions in which the same algorithm works on a wide range of data types, whereas *generic* refers to function

[4] A term like *categorical polymorphism* seems to be misleading, especially since we prefer the word type class instead of category.

declarations in the language which are resolved by different pieces of code.

However, a clear distinction can only be made if there is an untyped language to which the typed language is reduced.[5] On the other-hand if typing information is used by the run-time system it does not seem to be possible to have such a distinction. So in the book by Aho, et al. [Ahox86] no distinction is made between these terms.

Nevertheless, we will sometimes use these terms with the flavor as is given in [Fode83] when it will be clear how the language constructs in discussion can be translated into untyped ones.

### Coercions

We will assume that we have a mechanism in the language to declare some functions between types to be *coercions*, i. e. conversion functions which are automatically inserted by the system if necessary.

The usage of this terminology seems to be more or less standard, as the definition in the book by Aho, et al. [Ahox86, p. 359] shows:

> Conversion from one type to another is said to be *implicit* if it is to be done automatically by the compiler. Implicit type conversions, also called *coercions*, are limited in many languages to situations where no information is lost in principle;

The definitions in the glossary of the book on Axiom [Jenk92] is quite similar:

> **coercion**
> an automatic transformation of an object of one *type* to an object of a similar or desired target type. In the interpreter, coercions and *retractions* are done automatically by the interpreter when a type mismatch occurs. Compare **conversion**.
>
> **conversion**
> the transformation of an object of one *type* to one of another type. Conversions that can be performed automatically by the interpreter are called *coercions*. These happen when the interpreter encounters a type mismatch and a similar or declared target type is needed. In general, the user must use the infix operation "::" to cause this transformation.

However, there are some issues which have to be clarified. In the following a *coercion* will always be a *total function*. Although we will see that it is desirable to have injective coercions ("no information is lost in principle") we will not require that coercions are injective by the definition of the term.

We will use the term *retraction* for non-total conversion functions. Our usage of this term is more general than the one in Axiom:

> **retraction**
> to move an object in a parameterized domain back to the underlying domain, for example to move the object 7 from a "fraction of integers" (domain Fraction Integer) to "the integers" (domain Integer).

In several papers — e. g. [Fuhx90], [Mitc91] — the term *subtype* is used if there is a coercion from one type (the "subtype") into another type (the "supertype"). Since the term "subtype"

[5] This is the case for typed-functional programming languages which are usually translated into the untyped lambda-calculus. It can also be put in a precise form that the lambda-calculus is untyped.

has several other meanings in the literature, we will avoid it. Only in our notation we will be close to that terminology and will write $t_1 \trianglelefteq t_2$ if there is a coercion $\phi : t_1 \longrightarrow t_2$.

### 8.1.2 General Notation

As usual we will use *"iff"* for *"if, and only if"*.

The non-negative integers will be denoted by $\mathbb{N}$. The integers will be denoted by $\mathbb{Z}$ and the rationals by $\mathbb{Q}$. For $n \in \mathbb{N}$ we will denote the integers modulo $n$ by $\mathbb{Z}_n$. We will use these symbols both for the algebras (of the usual signatures) and the underlying sets. Since we use these ambiguous notations only in parts exclusively written for human beings and not for machines, there will not be any problems. Nevertheless, a major part of this thesis will deal with problems which arise from ambiguities which mathematicians usually can resolve easily. We will show how some of them can be treated in a clean formal way accessible to machines, sometimes causing computationally hard problems.

The set of strings over a set $L$ — i. e. the set of finite sequences of elements of $L$ — will be $L^*$, where $\varepsilon$ is the empty string.

The length of a string $s \in L^*$ will be denoted by $|s|$. We will also denote the cardinality of a set $A$ by $|A|$.

### 8.1.3 Partial Orders and Quasi-Lattices

**Definition 1. (Preorder).** *A binary relation which is reflexive and transitive is a* preorder.

A preorder which is also antisymmetric is a partial order.

**Definition 2.** *Let $\langle M, \leq \rangle$ be a partially ordered set. Then $c \in M$ is a* common lower bound *of $a$ and $b$ ($a, b \in M$) if $c \leq a$ and $c \leq b$.*

*Moreover, $c \in M$ is a* common upper bound *of $a$ and $b$ if $a \leq c$ and $b \leq c$.*

**Definition 3.** *Let $\langle M, \leq \rangle$ be a partially ordered set. Then $c \in M$ is called the* infimum *of $a$ and $b$ ($a, b \in M$) if $c$ is a lower bound of $a$ and $b$ and*

$$\forall d \in M : d \leq a \text{ and } d \leq b \Longrightarrow d \leq c.$$

*Furthermore, $c$ is called the* supremum *of $a$ and $b$ if it is an upper bound of $a$ and $b$ and*

$$\forall d \in M : a \leq d \text{ and } b \leq d \Longrightarrow c \leq d.$$

It is easy to verify that infima and suprema are unique if they exist. By induction the infimum and the supremum of any finite subset of a partially ordered set $\langle M, \leq \rangle$ can be defined.

**Definition 4.** *A partially ordered set $\langle M, \leq \rangle$ is a* lower quasi-lattice *if for any $a, b \in M$ $a$ and $b$ have an infimum whenever they have a common lower bound. It is a* lower semi-lattice *if any $a, b \in M$ have an infimum.*

*A partially ordered set $\langle M, \leq \rangle$ is an* upper quasi-lattice *if for for any $a, b \in M$ $a$ and $b$ have a supremum whenever they have a common upper bound. It is an* upper semi-lattice *if any $a, b \in M$ have a supremum.*

*A partially ordered set $\langle M, \leq \rangle$ is a* quasi-lattice *if it is an upper and a lower quasi-lattice. It is a* lattice *if it is both a upper and a lower semi-lattice.*

**Definition 5. (Free Lower Semi-Lattices)** Let $\langle M, \leq \rangle$ be a partially ordered set. The *free lower semi-lattice on* $\langle M, \leq \rangle$ is the following partially ordered set $\langle F, \preceq \rangle$:

1. $F$ is the set of all non-empty subsets of $M$ whose elements are pairwise incomparable with respect to $\leq$.
2. If $S_1, S_2 \in F$ then

$$S_1 \preceq S_2 \quad \Longleftrightarrow \quad \forall s_2 \in S_2 \, \exists s_1 \in S_1 \, . \, s_1 \leq s_2.$$

**Lemma 1. (Free Lower Semi-Lattices)** *Let $\langle M, \leq \rangle$ be a partially ordered set. Then the free lower semi-lattice on $\langle M, \leq \rangle$ is a lower semi-lattice.*

**Proof** *Let $S_1, S_2 \in F$ be arbitrary. Since $S_1 \in F$ and $S_2 \in F$ the chains in $S_1 \cup S_2$ with respect to $\leq$ have length at most 2. Let*

$$H = \{d \in S_1 \cup S_2 \mid \exists s \in S_1 \cup S_2 \, . \, s < d\}$$

*and*

$$\overline{S} = (S_1 \cup S_2) - H.$$

Since $\overline{S}$ is not empty and contains only incomparable elements by construction we have $\overline{S} \in F$.

We claim that $\overline{S}$ is the infimum of $S_1$ and $S_2$.

We have $\overline{S} \preceq S_1$ because for any $s \in S_1$ either $s \in \overline{S}$ or there is a $s' \in \overline{S}$ such that $s' < s$. Similarly $\overline{S} \preceq S_2$.

Let $L \in F$ be a common lower bound of $S_1$ and $S_2$ with respect to $\preceq$, i. e. $L \preceq S_1$ and $L \preceq S_2$. Then for any $s \in S_1 \cup S_2$ there is an $l \in L$ such that $l \leq s$. Since

$$\overline{S} \subseteq S_1 \cup S_2$$

we thus have $L \preceq \overline{S}$ by the definition of $\preceq$. □

**Remark** The statement given in [Nipk91, p. 9] that the union of two sets of incomparable elements is a set of incomparable elements and is the infimum of these sets with respect to the ordering given in Def. 5 is false in general. The proof of Lemma 1 shows the correct construction.

**Remark** If we define semi-lattices algebraically (see e. g. [Grae79, § 6]), then the free lower semi-lattice on $\langle M, \leq \rangle$ is indeed a free semi-lattice.

**Lemma 2.** *Let $\langle M, \leq \rangle$ be a* finite *partially ordered set. Then $\langle M, \leq \rangle$ is a lower quasi-lattice iff it is an upper quasi-lattice.*

**Proof** Let $\langle M, \leq \rangle$ be a lower quasi-lattice in which $a$ and $b$ have a common upper bound. We have to show that $a$ and $b$ have a supremum. Since the set

$$I = \{c \in M : a \leq c \text{ and } b \leq c\}$$

is nonempty and finite and $\langle M, \leq \rangle$ is a lower quasi-lattice the infimum $c$ of $I$ exists. Clearly $c$ is the supremum of $a$ and $b$.

The other direction is shown analogously. □

**Lemma 3.** *Let $\langle M, \leq \rangle$ be a finite partially ordered set. Then $\langle M, \leq \rangle$ is not a quasi-lattice iff there are $a, b, c, d \in M$ such that*

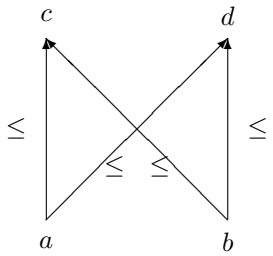



Figure 8.1: Ad Lemma 3

1. $a \leq c$ *and* $a \leq d$,
2. $b \leq c$ *and* $b \leq d$,
3. $a \not\leq b$ *and* $b \not\leq a$,
4. $c \not\leq d$ *and* $d \not\leq c$,
5. *there is no* $e \in M$ *which is a common upper bound of* $a$ *and* $b$ *and a common lower bound of* $c$ *and* $d$.

**Proof** Assume $\langle M, \leq \rangle$ is a finite partially ordered set having elements $a, b, c, d \in M$ that satisfy the conditions of the lemma. Since $a$ and $b$ have a common upper bound, we are done if we can show that they do not have a supremum. Assume towards a contradiction they had a supremum $e$. Since $c$ and $d$ are common upper bounds of $a$ and $b$ and $e$ is the supremum of $a$ and $b$, we had $e \leq c$ and $e \leq d$, a contradiction to condition 5 of the lemma.

Now let $\langle M, \leq \rangle$ be a finite partially ordered set which is not a quasi-lattice. Then there are $a, b \in M$ which have a common upper bound $c$ but not a supremum. Since $M$ is finite we can assume w. l. o. g. that there is no $c' \leq c$ which is also a common upper bound of $a$ and $b$ (if there is one, take $c'$ instead of $c$). Since $a$ and $b$ have no supremum, there is a common upper bound $d$ of $a$ and $b$ such that $d \not\leq c$ and $c \not\leq d$. These elements $a, b, c, d$ satisfy the conditions of the lemma. □

### 8.1.4 Order-Sorted Algebras

There is a growing literature on order-sorted algebras. Some comprehensive sources are the thesis of Schmidt-Schauß [Schm89], the survey by Smolka, et al. [Smol89], and the articles by Goguen & Meseguer [Gogu89] and by Waldmann [Wald92]. In [Como90] Comon shows that an order-sorted signature can be viewed as a finite bottom-up tree automaton.

**Definition 6. (Order-Sorted Signature)** *An* order-sorted signature *is a triple* $(S, \leq, \Sigma)$, *where* $S$ *is a set of sorts,* $\leq$ *a preorder on* $S$, *and* $\Sigma$ *a family*

$$\{\Sigma_{\omega,\sigma} \mid \omega \in S^*,\ \sigma \in S\}$$

*of not necessarily disjoint sets of operator symbols.*

*If* $S$ *and* $\Sigma$ *are finite, the signature is called finite.*

*For notational convenience, we often write* $f : (\omega)\sigma$ *instead of* $f \in \Sigma_{\omega,\sigma}$*;* $(\omega)\sigma$ *is called an* arity *and* $f : (\omega)\sigma$ *a* declaration. *The signature* $(S, \leq, \Sigma)$ *is often identified with* $\Sigma$*. If* $|\omega| = n$ *then* $f$ *is called a* $n$*-ary operator symbol.* 0*-ary operator symbols are* constant symbols. *As in [?] we will assume in the following that for any* $f$ *there is only a single* $n \in \mathbb{N}$ *such that* $f$ *is a* $n$*-ary operator symbol.*

*An* $\sigma$*-sorted variable set is a family*

$$V = \{V_\sigma \mid \sigma \in S\}$$

*of disjoint, nonempty sets. For* $x \in V_\sigma$ *we also write* $x : \sigma$ *or* $x_\sigma$*.*

In [Gogu89] the following monotonicity condition must be fulfilled by any order-sorted signature.

**Definition 7.** *An order-sorted signature* $(S, \leq, \Sigma)$ *fulfills the* monotonicity condition*, if*

$$f \in \Sigma_{\omega_1,\sigma_1} \cap \Sigma_{\omega_2,\sigma_2} \text{ and } \omega_1 \leq \omega_2 \text{ imply } \sigma_1 \leq \sigma_2.$$

Notice that the monotonicity condition excludes multiple declarations of constants. This is one of the reasons why we will not assume in general that the order-sorted signatures we will deal with will fulfill the monotonicity condition.

**Definition 8. (Order-Sorted Terms)** *The set of* order-sorted terms *of sort* $\sigma$ *freely generated by* $V$*,* $T_\Sigma(V)_\sigma$*, is the least set satisfying*

- if $x \in V_{\sigma'}$ and $\sigma' \leq \sigma$, then $x \in T_\Sigma(V)_\sigma$
- if $f \in \Sigma_{\omega,\sigma'}$, $\omega = \sigma_1 \ldots \sigma_n$, $\sigma' \leq \sigma$ and $t_i \in T_\Sigma(V)_{\sigma_i}$ then $f(t_1, \ldots, t_n) \in T_\Sigma(V)_\sigma$.

If $t \in T_\Sigma(V)_\sigma$ we will also write $t : \sigma$.

In contrast to sort-free terms and variables, order-sorted variables and terms always have a sort. Terms must be sort-correct, that is, subterms of a compound term must be of an appropriate sort as required by the arities of the term's operator symbol.

Note that an operator symbol may have not just one arity (as in classical homogeneous or heterogeneous term algebras), but may have *several* arities. As a consequence, each term may have several sorts.

The set of all order-sorted terms over $\Sigma$ freely generated by $V$ will be denoted by

$$T_\Sigma(V) := \bigcup_{\sigma \in S} T_\Sigma(V)_\sigma.$$

The set of all *ground terms* over $\Sigma$ is $T_\Sigma := T_\Sigma(\{\})$.

**Definition 9. (Regularity)** *A signature is* regular*, if the subsort preorder of* $\Sigma$ *is anti-symmetric, and if each term* $t \in T_\Sigma(V)$ *has a least sort.*

The following theorem shows that it is decidable for finite signatures whose subsort preorders are anti-symmetric if a signature is regular.

**Theorem 1.** *A signature* $(S, \leq, \Sigma)$ *whose subsort preorder is anti-symmetric is regular iff for every* $f \in \Sigma$ *and* $\omega \in S^*$ *the set*

$$\{\sigma \mid \exists \omega' \geq \omega : f \in \Sigma_{\omega',\sigma}\}$$

*either is empty or contains a least element.*

**Proof** See [Smol89]. □

As an example of a simple non-regular signature, consider

$$(\{\sigma_0, \sigma_1, \sigma_2\},\ \{\sigma_1 \leq \sigma_0,\ \sigma_2 \leq \sigma_0\},\ \Sigma_{\varepsilon,\sigma_1} = a, \Sigma_{\varepsilon,\sigma_2} = a).$$

The constant $a$ has two sorts which are incomparable, hence it does not have a minimal sort.

**Definition 10.** *The* complexity *of a term $t \in T_\Sigma(V)$, $\mathrm{com}(t)$ is inductively defined as follows:*

- $\mathrm{com}(t) = 1$, *if $t \in V_\sigma$ or $t \in \Sigma_{\epsilon,\sigma}$ for some $\sigma \in S$,*
- *if $f \in \Sigma_{\omega,\sigma'}$, $\omega = \sigma_1 \cdots \sigma_n$, and $t_i \in T_\Sigma(V)_{\sigma_i}$ then*

$$\mathrm{com}(f(t_1, \ldots, t_n)) = \max(\mathrm{com}(t_1), \ldots, \mathrm{com}(t_n)) + 1.$$

**Definition 11.** *A* substitution *$\theta$ from a variable set $Y$ into the term algebra $T_\Sigma(V)$ is a mapping from $Y$ to $T_\Sigma(V)$, which additionally satisfies $\theta(x) \in T_\Sigma(V)_\sigma$ if $x \in V_\sigma$ (that is, substitutions must be sort-correct). As usual, substitutions are extended canonically to $T_\Sigma(V)$. If $Y = \{x_1, \ldots, x_n\}$ we write $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. If $\theta = \{x_1 \mapsto t_1\}$ and $t \in T_\Sigma(V)$, then we will write $t[t_1/x_1]$ for $\theta(t)$. If, for $t, t' \in T_\Sigma(V)$, there is a substitution $\theta$ such that $t' = \theta(t)$, then $t'$ is called an* instance *of $t$. Similarly, a substitution $\theta'$ is called an instance of a substitution $\theta$ with respect to a set of variables $W$, written $\theta \succeq \theta'[W]$, if there is a substitution $\gamma$ such that $\theta'(x) = \gamma(\theta(x))$ for all $x \in W$.*

**Definition 12.** *A* unifier *of a set of equations $\Gamma$ is a substitution $\theta$ such that $\theta(s) = \theta(t)$ for all equations $s =^? t$ in $\Gamma$. A set of unifiers $U$ of $\Gamma$ is called* complete *(and denoted by CSU), if for every unifier $\theta'$ of $\Gamma$ there exists $\theta \in U$ such that $\theta'$ is an instance of $\theta$ with respect to the variables in $\Gamma$. As usual, a signature is called* unitary (unifying), *if for all equation sets $\Gamma$ there is a complete set of unifiers containing at most one element; it is called* finitary (unifying), *if there is always a finite and complete set of unifiers.*

For non-regular signatures, unifications can be infinitary, even if the signature is finite (see e. g. [Smol89, p. 326], [Wald92, p. 26]).

**Theorem 2. (Schmidt-Schauß)** *In finite and regular signatures, finite sets of equations have finite, complete, and effectively computable sets of unifiers.*

**Proof** See [Smol89, Theorem 15]. □

**Definition 13.** *A signature $(S, \leq, \Sigma)$ is* downward complete *if any two sorts have either no lower bound or an infimum, and* coregular *if for any operator symbol $f$ and any sort $\sigma \in S$ the set*

$$D(f, \sigma) = \{\omega \mid \exists \sigma' \in S\ .\ f : (\omega)\sigma' \wedge \sigma' \leq \sigma\}$$

*either is empty or has a greatest element.*

**Definition 14.** *Let $(S, \leq, \Sigma)$ be an order-sorted signature. It is* injective *if for any operator symbol $f$ the following condition holds:*

$$f : (\omega)\sigma \text{ and } f : (\omega')\sigma \quad \textit{imply} \quad \omega = \omega'.$$

*It is* subsort reflecting *if for any operator symbol $f$ the following condition holds:*

$$f : (\omega')\sigma' \text{ and } \sigma' \leq \sigma \quad \text{imply} \quad f : (\omega)\sigma \text{ for some } \omega \geq \omega'.$$

**Theorem 3.** *Every finite, regular, coregular, and downward complete signature is unitary unifying.*

**Proof** See [Smol89, Theorem 17]. □

**Corollary 3A.** *Every finite, regular, downward complete, injective and subsort reflecting signature is unitary unifying.*

### 8.1.5 Category Theory

We will recall some basic definitions from category theory which can be found in many books on the topic. Some classical textbooks are [Macl91], [Schu72]. A more recent textbook is [Frey90]. In [Ryde88] computational aspects are elaborated. The basic concepts of category theory can also be found in several books which use category theory as a tool for computer science, e. g. in [Ehri85].

**Definition 15. (Category)** *A* category $\mathcal{C}$ *consists of a class of* objects $\mathrm{Obj}_{\mathcal{C}}$, *for each pair* $(A, B) \in \mathrm{Obj}_{\mathcal{C}} \times \mathrm{Obj}_{\mathcal{C}}$ *a set* $\mathrm{Mor}_{\mathcal{C}}(\mathrm{A}, \mathrm{B})$ *of* morphisms *(or* arrows*), written* $f : A \longrightarrow B$ *for* $f \in \mathrm{Mor}_{\mathcal{C}}(\mathrm{A}, \mathrm{B})$, *and a* composition

$$\begin{aligned} &\circ : \mathrm{Mor}_{\mathcal{C}}(\mathrm{A}, \mathrm{B}) \times \mathrm{Mor}_{\mathcal{C}}(\mathrm{B}, \mathrm{C}) \longrightarrow \mathrm{Mor}_{\mathcal{C}}(\mathrm{A}, \mathrm{C}) \\ &(f : A \longrightarrow B,\ g : B \longrightarrow C) \mapsto (g \circ f : A \longrightarrow C) \end{aligned}$$

*(more precisely a family of functions* $\circ_{A,B,C}$ *for all objects* $A, B, C$*) such that the following axioms are satisfied:*

1. $(h \circ g) \circ f = h \circ (g \circ f)$ (associativity)
   *for all morphisms* $f, g, h$, *if at least one side is defined.*
2. *For each object* $A \in \mathrm{Obj}_{\mathcal{C}}$ *there is a morphism* $\mathrm{id}_A \in \mathrm{Mor}_{\mathcal{C}}(\mathrm{A}, \mathrm{A})$, *called* identity of $A$, *such that we have for all* $f : A \longrightarrow B$ *and* $g : C \longrightarrow A$ *with* $B, C \in \mathrm{Obj}_{\mathcal{C}}$
   $f \circ \mathrm{id}_A = f$ *and* $\mathrm{id}_A \circ g = g$ (identity).

*Frequently we will write*

$$A \xrightarrow{f} B$$

*instead of*

$$f : A \longrightarrow B.$$

**Definition 16. (Opposite Category)** *Let* $\mathcal{C}$ *be a category. Then* $\mathcal{C}^{\mathrm{op}}$, *the* opposite category *of* $\mathcal{C}$, *is the category which is defined by*

1. $\mathrm{Obj}_{\mathcal{C}^{\mathrm{op}}} = \mathrm{Obj}_{\mathcal{C}}$,
2. $\mathrm{Mor}_{\mathcal{C}^{\mathrm{op}}}(\mathrm{A}, \mathrm{B}) = \mathrm{Mor}_{\mathcal{C}}(\mathrm{B}, \mathrm{A})$.

*Sometimes we will call* $\mathcal{C}^{\mathrm{op}}$ *the* dual category *of* $\mathcal{C}$.

Clearly, $(\mathcal{C}^{\mathrm{op}})^{\mathrm{op}} = \mathcal{C}$.

For any categories $\mathcal{C}$ and $\mathcal{D}$ we will write $\mathcal{C} \times \mathcal{D}$ for the category which is defined by

1. $\mathrm{Obj}_{\mathcal{C} \times \mathcal{D}} = \mathrm{Obj}_{\mathcal{C}} \times \mathrm{Obj}_{\mathcal{D}}$,
2. $\mathrm{Mor}_{\mathcal{C} \times \mathcal{D}}((\mathrm{A}, \mathrm{A}'), (\mathrm{B}, \mathrm{B}')) = \mathrm{Mor}_{\mathcal{C}}(\mathrm{A}, \mathrm{B}) \times \mathrm{Mor}_{\mathcal{D}}(\mathrm{A}', \mathrm{B}')$,

where the symbol $\times$ on the right hand side of the equations denotes the usual set theoretic Cartesian product.

Since $\times$ is associative, we will write unambiguously $\mathcal{C}_1 \times \cdots \times \mathcal{C}_n$ for an $n$-fold iteration. Moreover,

$$\mathcal{C}^n = \underbrace{\mathcal{C} \times \cdots \times \mathcal{C}}_{n}.$$

**Definition 17. (Functors)** *Let $\mathcal{C}$ and $\mathcal{D}$ be categories. A mapping $F : \mathcal{C} \longrightarrow \mathcal{D}$ is called* functor, *if $F$ assigns to each object $A$ in $\mathcal{C}$ an object $F(A)$ in $\mathcal{D}$ and to each morphism $f : A \longrightarrow B$ in $\mathcal{C}$ a morphism $F(f) : F(A) \longrightarrow F(B)$ in $\mathcal{D}$ such that the following axioms are satisfied:*

1. *$F(g \circ f) = F(g) \circ F(f)$ for all $g \circ f$ in $\mathcal{C}$,*
2. *$F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$ for all objects $A$ in $\mathcal{C}$.*

*The composition of two functors $F : \mathcal{C} \longrightarrow \mathcal{D}$ and $G : \mathcal{D} \longrightarrow \mathcal{E}$ is defined by*

$$G \circ F(A) = G(F(A))$$

*and*

$$G \circ F(f) = G(F(f))$$

*for objects and morphisms respectively leading to the* composite functor $G \circ F : \mathcal{C} \longrightarrow \mathcal{E}$.

*The* identical functor $\mathrm{ID}_{\mathcal{C}} : \mathcal{C} \longrightarrow \mathcal{C}$ *is defined by* $\mathrm{ID}_{\mathcal{C}}(A) = A$ *and* $\mathrm{ID}_{\mathcal{C}}(f) = f$.

*A functor $F : \mathcal{C} \longrightarrow \mathcal{D}$ is also called a* covariant functor *from $\mathcal{C}$ into $\mathcal{D}$. A functor $F : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{D}$ is called a* contravariant functor *from $\mathcal{C}$ into $\mathcal{D}$.*

**Definition 18. (Natural Transformations)** *Let $S, T : \mathcal{C} \longrightarrow \mathcal{D}$ be functors. A* natural transformation *$\tau : S \longrightarrow T$ is a mapping which assigns to any object $A$ in $\mathcal{C}$ an arrow $\tau_A = \tau(A) : S(A) \longrightarrow T(A)$ such that for any arrow $f : A \longrightarrow B$ in $\mathcal{C}$ the diagram*

$$\begin{array}{ccc} S(A) & \xrightarrow{\tau(A)} & T(A) \\ {\scriptstyle S(f)}\downarrow & & \downarrow{\scriptstyle T(f)} \\ S(B) & \xrightarrow[\tau(B)]{} & T(B) \end{array}$$

*is commutative.*

**Defintion 19. (Initial Objects)** *Let $\mathcal{C}$ be a category. An object $I \in \mathrm{Obj}_{\mathcal{C}}$ is* initial in $\mathcal{C}$ *if for any object $A \in \mathcal{C}$ there is a unique morphism $f \in \mathrm{Mor}_{\mathcal{C}}(\mathrm{I}, \mathrm{A})$. If the category $\mathcal{C}$ is clear from the context, then it is simply said that $I$ is an* initial object.

If an initial object exists in a category, it is uniquely determined up to isomorphism.

### 8.1.6 The Type System of Axiom

The type system of Axiom consists of three levels:

1. elements,
2. domains,
3. categories.

The elements belong to domains, which correspond to types in the traditional programming terminology.

Domains are built by *domain constructors*, which are functions having the following sort of parameters: elements, or domains of a certain category. Any domain constructor has a *category assertion* part which asserts that for any possible parameters of the domain constructor the constructed domain belongs to the categories given in it.

### Categories

Also categories are built by category constructors which are functions having elements or domains as parameters.

An important subclass is built by the categories which are built by category constructors having no parameters. They are called the *basic algebra hierarchy* in [Jenk92] and consist up to now of 46 categories.

As is stated in [Jenk92, p. 524] the case of elements as parameters of category constructors is rare.

In the definition of a category there is always a part in which the categories are given the defined category extends.[6]

An important component of the definition of a category is the documentation. There is even a special syntax for comments serving as a documentation in contrast to other kinds of comments. The importance of having a language support for the documentation as well as for the implementation of an algorithm is also clearly elaborated in the design of the algorithm description language ALDES [Loos72], [Loos76], [Loos92], and in the implementation of the SAC-2 library (see e. g. [Coll90], [Buen91]).

The *axioms* which a member of a category has to fulfill are stated in the comment only and there is no mechanism for an automated verification provided yet. There is a mechanism to declare some equationally definable axioms as so called *attributes* which can be used explicitly in the language. However, the attributes can be used only directly. A theorem proving component is not included in the system.

Some operations in a category definition can have *default* declarations, i. e. algorithms for algorithmically definable operations can be given. These default declarations can be overwritten by special algorithms in any instance of a category.

There are two syntactic declarations which reduce the number of category declarations which have to be given considerably.

Using the keyword `Join` a category is defined which has all operations and properties of the categories given as arguments to `Join`.

Instead of defining different categories $\mathcal{C}_1$ and $\mathcal{C}_2$ and to declare that $\mathcal{C}_2$ extends $\mathcal{C}_1$ it is possible to define $\mathcal{C}_1$ and to use the so called *conditional phrase* `has` in the definition of $\mathcal{C}_1$ to give the additional properties of $\mathcal{C}_2$.

### Coercions

In Axiom it is possible to have coercions between domains. Syntactically, an overloaded operator symbol `coerce` is used for the definition of the coercion functions. There seems to be no restriction on the functions which can be coercions. So also partial functions can be coercions in Axiom in contrary to the usage of the term in this chapter.

[6] The category which is extended by all other categories and which does not extend any other category is predefined and is called `Type`.

```
++ the class of all multiplicative semigroups
++ Axioms
++ .  associative("*":($,$)->$)            ++ (x*y)*z = x*(y*z)
++ Common Additional Axioms
++ .  commutative("*":($,$)->$)            ++ x*y = y*x
SemiGroup(): Category == SetCategory with
    --operations
      "*": ($,$) -> $
      "**": ($,PositiveInteger) -> $

    add
      import RepeatedSquaring($)
      x:$ ** n:PositiveInteger == expt(x,n)
```

Figure 8.2: An example of a category definition in Axiom

## 8.2 Type Classes

In the main part of this section we will deal with language constructs which correspond to *categories* of Axiom obeying the restriction of having no parameters. In Sec. 8.2.5 we will discuss the case of categories with parameters.

The momentarily occurring examples of such categories are given as the "basic algebra hierarchy" on the inner cover page of the book on Axiom [Jenk92]. They consist of 46 categories. The maximal length of a chain in the induced partial order is 15.

These categories correspond to type classes of Haskell, cf. Fig. 8.4 and Fig. 8.3. We will often use the term *type class* — which seems to be preferable — instead of non-parameterized category.

In [Webe92b, Appendix A] the author has shown that almost all of the examples of types occurring in the specifications of the SAC-2 library (see e. g. [Coll90], [Buch93]) can be structured by using the language construct of type classes.

We will also assume that all domain constructors have only domains as parameters, and not elements of other domains. We will discuss the extension of having elements of domains as parameters in Sec. 8.4.2.

### 8.2.1 Types as Terms of an Order-Sorted Signature

The idea of describing the types of a computer algebra system as terms of an order-sorted signature can also be found in the work of Rector [Rect89] and Comon, et al. [Como91]. The idea of describing the type system of Haskell using order-sorted terms is due to Nipkow and Snelting [Nipk91].

However, the combination of ideas found in these papers is new and gives a solution to an important class of type inference problems occurring in computer algebra.

In the following a *type* will just be an element of the set of all order-sorted terms over a signature $(S, \leq, \Sigma)$ freely generated by some family of infinite sets $V = \{V_\sigma \mid \sigma \in S\}$.

The sorts correspond to the non-parameterized categories, the basic algebra hierarchy. The order on the sorts reflects the inheritance mechanism of categories.

```
class  (Eq a) => Ord a  where
  (<), (<=), (>=), (>):: a -> a -> Bool
  max, min            :: a -> a -> a

  x <  y              =  x <= y && x /= y
  x >= y              =  y <= x
  x >  y              =  y <  x

  -- The following default methods are appropriate for partial orders.
  -- Note that the second guards in each function can be replaced
  -- by "otherwise" and the error cases, eliminated for total orders.
  max x y | x >= y    =  x
          | y >= x    =  y
          |otherwise  =  error "max{PreludeCore}: no ordering relation"
  min x y | x <= y    =  x
          | y <= x    =  y
          |otherwise  =  error "min{PreludeCore}: no ordering relation"
```

Figure 8.3: Definition of partially ordered sets in the Haskell standard prelude

```
++ Totally ordered sets
++ Axioms
++ . a<b or a=b or b<a (and only one of these!)
++ . a<b and b<c => a<c
OrderedSet(): Category == SetCategory with
  --operations
    "<": ($,$) -> Boolean    ++ The (strict) comparison operator
    max: ($,$) -> $          ++ The maximum of two objects
    min: ($,$) -> $          ++ The minimum of two objects
  add
  --declarations
    x,y: $
  --definitions
  -- These really ought to become some sort of macro
    max(x,y) ==
      x > y => x
      y
    min(x,y) ==
      x > y => y
      x
```

Figure 8.4: Definition of totally ordered sets in Axiom

The sets $V_\sigma$ are the sets of *type variables*.

A type denoted be a ground term is called a *ground type*, a non-ground type is called a *polymorphic type*. Polymorphic types correspond to the *modemaps* of Axiom.

A type denoted by a constant symbol will be called a *base type*. So base types correspond to domains built by domain constructors without parameters. (Typical examples are `integer`, `boolean`, ...)

The non-constant operator symbols are called *type constructors*. The domain constructors of Axiom which have only domains as parameters can be described by type constructors.

We will use

```
list : (any)any
list : (ordered_set)ordered_set
UP : (commutative_ring symbol)commutative_ring
UP : (integral_domain symbol)integral_domain
FF : (integral_domain)field
```

as typical examples, where `UP` builds univariate polynomials in a specified indeterminate of a commutative ring, and `FF` the field of fractions of an integral domain.

Notice the use of multiple declarations, which can be achieved in Axiom using the conditional phrase `has`.

In the following we will sometimes assume that we have a *semantics* for the *ground types* which satisfies the following conditions:

- The ground types correspond to mathematical objects in the sense of universal algebra or model theory (A comprehensive reference for universal algebra is [Grae79], for model theory [Chan90]).
- Functions between ground types are set theoretical functions. If we say that two functions $f, g : t_1 \longrightarrow t_2$ are equal ($f = g$) we mean equality between them as set theoretic objects.

Since we only need a set theoretic semantics for *ground types* and functions between ground types, the obvious interpretations of the types as set theoretic objects will do.[7]

Of course, equality between two functions will be in general an undecidable property, but this will not be of importance in the following discussion, since we will always give some particular reasoning for the equality of two functions between two types.

We will also deal with polymorphic types in the following. However, it will not be necessary to have a formal semantics for the polymorphic types in the cases we will use them. Giving a semantics to polymorphic types can be quite difficult. So the one given in [Como91] applies to fewer cases than the ones we are interested in. In general, it is possible that no "set-theoretic semantics" can be given to polymorphic types, as was shown by Reynolds [Reyn84] for the objects of the second-order polymorphic lambda-calculus.

### Properties of the Order-Sorted Signature of Types

The possibility to have multiple declarations of type constructors is used in Axiom frequently. Syntactically it is achieved by a conditional phrase involving `has`.[8]

Also constant symbols, i. e. base types, have usually multiple declarations, e. g. it is useful to declare `integer` to be an integral_domain and an ordered_set. So the monotonicity condition cannot be assumed in general. However, for the purposes of type inference (see below) this condition is not needed.

[7] All objects corresponding to ground types one is interested in computer algebra can be given such a set theoretic interpretation. In other areas, e. g. in the context of the lambda calculus [Bare84] this is not always the case. Nevertheless, this is not a real problem for our work, since our approach is primarily concerned with the situation arising in computer algebra.

[8] In Axiom conditional phrases are used also for other purposes. So it might be useful to use different syntactic concepts instead of one.

As is shown in [Nipk91, Sec. 5] it can be assumed that the signature is regular[9] and downward complete if one allows to form the "conjunction" $\sigma_1 \wedge \sigma_2$ of sorts $\sigma_1$ and $\sigma_2$. This conjunction has to fulfill the following conditions:

1. $\sigma_1 \wedge \sigma_2$ has to be the meet of $\sigma_1$ and $\sigma_2$ in the free lower semi-lattice on the partially ordered set $\langle S, \leq \rangle$ (cf. Def. 5).
2. If a type constructor $\chi$ has declarations $\chi : (\gamma_1 \cdots \gamma_n)\gamma$ and $\chi : (\delta_1 \cdots \delta_n)\delta$ then it also has a declaration
$$\chi : (\gamma_1 \wedge \delta_1 \ \cdots \gamma_n \wedge \delta_n)\gamma \wedge \delta.$$

Using `Join` there is a possibility to form such conjunctions of sort having the required properties in Axiom.

**Remark** Maybe the choice of the name `Join` in Axiom is somewhat misleading. Although the `Join` of two categories gives a category having the union of their operations, this category is nevertheless corresponding to the *meet* of the corresponding sorts in the lower semi-lattice of sorts of the order-sorted signature of types. We cannot simply reverse the order on the sorts. If a type belongs to the join of two categories $\mathcal{A}$ and $\mathcal{B}$ we can conclude that it belongs to $\mathcal{A}$ (or $\mathcal{B}$) but not vice versa!

For the purpose of type inference it would be nice if the signature is unitary unifying. This is the case for regular and downward complete signatures if they are also *coregular*. However, we do not know whether a restriction implying coregularity is reasonable in the context of a computer algebra system.

Nipkow and Snelting [Nipk91] have argued that Haskell enforces that the order-sorted signatures are injective and subsort reflecting which also imply that the signature is unitary unifying.

An example of a declaration which would prohibit that the signature is *injective* is the following. Consider the type constructor `FF` building the field of fractions of an integral domain. Then the declarations

```
FF : (integral_domain)field
FF : (field)field
```

correctly reflect certain mathematical facts. Although it does not seem to be necessary in this example to have the second declaration we do not know whether there is an "algebraic" reason which implies that declarations violating injectivity are not necessary. So this point might deserve further investigations.

### Definition of Overloaded Functions

The formalism developed above is well suited to express the overloading which can be performed by category definitions.
A declaration such as

```
AbelianSemiGroup(): Category == SetCategory with
    --operations
      "+": ($,$) -> $            ++ x+y computes the sum of x and y
      "*": (PositiveInteger,$) -> $
```

[9] At least if the signature is finite.

would translate into

$$+ : \forall t_{\mathsf{AbelianSemiGroup}} \,.\, t_{\mathsf{AbelianSemiGroup}} \times t_{\mathsf{AbelianSemiGroup}} \longrightarrow t_{\mathsf{AbelianSemiGroup}},$$
$$* : \forall t_{\mathsf{AbelianSemiGroup}} \,.\, \mathtt{PositiveInteger} \times t_{\mathsf{AbelianSemiGroup}} \longrightarrow t_{\mathsf{AbelianSemiGroup}},$$

where $t_{\mathsf{AbelianSemiGroup}}$ is a type variable of sort AbelianSemiGroup. It is bounded by the universal quantifier which has to be read that $t_{\mathsf{AbelianSemiGroup}}$ may be instantiated by an arbitrary type of sort AbelianSemiGroup. This is just what we want. So the definition of categories resp. type classes can be seen as a syntactic mechanism to give such declarations of overloaded operators. The mechanism to declare that a category extends others can be simply modeled by the order relation on the sorts in the order-sorted algebra of types — if there are no parameters in category definitions.[10]

An advantage of the syntactic form of type classes declarations is certainly that the general declaration of the overloaded operators and possible *default declarations* are collected in one piece of code. This collection improves readability and makes clear which operators can have defaults and which cannot.

The value of default declarations may not be underestimated. They are a good way to support rapid prototyping and will become more important the bigger a system grows. They support the possibility to obtain algorithms over new structures quite easily. Since it is always possible to "overwrite" a default operation by a more special and efficient one their existence does not contradict the goal of having algorithms which are as efficient as possible.

In his thesis [Fode83] Foderaro distinguishes between an "operation centered" method and a "type centered" or "object-oriented" view of organizing data (cf. Fig. 8.5) and argues why the type-centered approach has to be preferred.

However, in our formalism these two views are essentially equivalent. There is a translations of a declaration of a type class — say Ring — and an instantiation of it — say with `integer` — with operations `integer_plus` and `integer_times` into declarations

$$+ : \forall t_{\mathsf{Ring}} \,.\, t_{\mathsf{Ring}} \times t_{\mathsf{Ring}} \longrightarrow t_{\mathsf{Ring}},$$
$$* : \forall t_{\mathsf{Ring}} \,.\, t_{\mathsf{Ring}} \times t_{\mathsf{Ring}} \longrightarrow t_{\mathsf{AbelianSemiGroup}},$$

whereas it can be deduced by a type inference algorithm that `integer_plus` has to be used for $+$ if $t_{\mathsf{Ring}}$ is instantiated with the type constant `integer`. We will present this inference algorithm in the next section.

### 8.2.2 Type Inference

In the following we will show that the type inference problem is decidable. We will sketch the proof which is due to Nipkow and Snelting [Nipk91] because of its importance also for computer algebra.

In Sec. 8.2.2 we will give some examples in which the Axiom type inference mechanism fails whereas in Haskell a type can be deduced.

[10] The inheritance mechanism is certainly convenient for such a large system as Axiom — as we have mentioned before, even the basic algebra hierarchy consists of 46 categories with chains of maximal length of 15 —, although it can be questioned whether it is really necessary, cf. [Chen92].

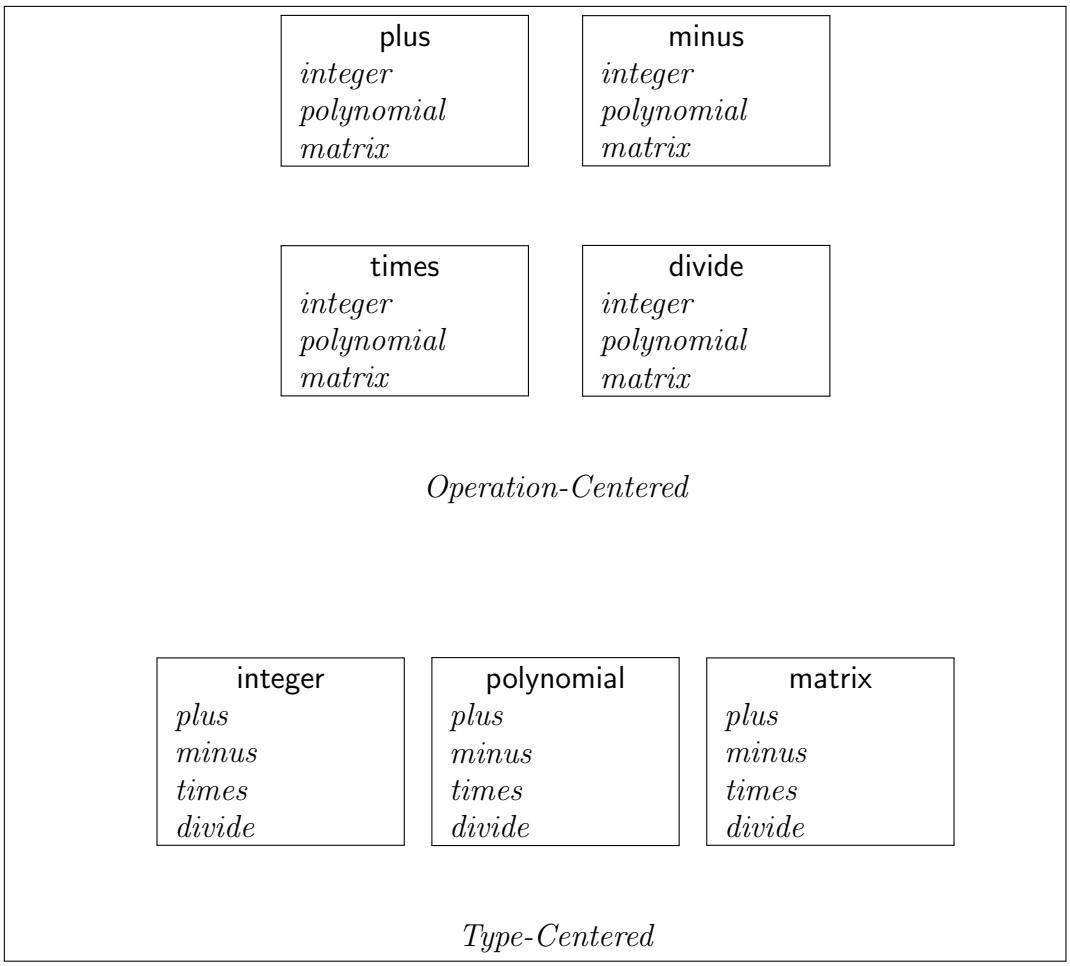



Figure 8.5: Some terminology from Foderaro's thesis

### Type Inference Rules of Mini-Haskell

In Fig. 8.6 the type inference rules for the language Mini-Haskell of Nipkow and Snelting [Nipk91] are given. This language includes the central typing concepts of Haskell but is well suited for theoretical investigations since it is very small. Many useful properties of an actual programming language can be seen as "syntactic sugar" for the purpose of the type inference problem.

Mini-Haskell can only handle unary functions. However, in this assumption there is no loss in generality. Since Mini-Haskell has higher order-functions, a function of type

$$\tau_1 \times \tau_2 \longrightarrow \tau_3$$

can be expressed by a function having type

$$\tau_1 \longrightarrow (\tau_2 \longrightarrow \tau_3),$$

a technique usually called *currying*.[11]

[11] After Haskell B. Curry who has used this technique in his work on Combinatory logic. Historically,

The language does not have explicit recursion or pattern matching. Although these are important properties of a programming language, there is no loss in generality in the type inference problem if we exclude them from the language. There are well known translations of pattern matching into expressions of the lambda-calculus, see e. g. [Jone87]. In principle, recursion can be expressed using fixpoint combinators which only requires to have certain appropriately typed functional constants (see e. g. [Leis87]).

**Remark** Having explicit recursion and some special typing rules for recursion gives the possibility to assign typing to some recursive programs which would be ill-typed otherwise (see e. g. [Kfou88], [Tiur90]). However, in some of these systems type inference becomes undecidable [Tiur90], [Kfou93].

**Remark** The so called "anonymous functions" in Axiom [Jenk92, Sec. 6.17] can simply be seen as $\lambda$-abstracted expressions. Since recursion can be expressed by the use of fixpoint combinators, also $\lambda$-expressions without names can be recursive,[12] in contrast to the remark in [Jenk92, p. 168]: "An anonymous function cannot be recursive: since it does not have a name, you cannot even call it within itself!"

In the following we will use the notation of Nipkow and Snelting [Nipk91] which has some syntactic differences to our standard notation but should be clear from the context. Since the type of functions between $\tau$ and $\tau'$ has a special role in the following there is a special notation for it and it is written as $\tau \longrightarrow \tau'$. The meta-variable $\chi$ ranges over type constructors, where it is assumed that a finite set of them is given (e. g. having `int`, `float`, `list`$(\alpha)$, `pair`$(\alpha, \beta)$ as members, as in [Nipk91]).

Formally, a typing hypothesis $A$ is a mapping from a finite set of variables to types. We will write

$$A + [x \mapsto \tau]$$

for the mapping which assigns $\tau$ to $x$ and is equal to $A$ on $\mathrm{dom}(A) - \{x\}$.[13] For signatures, the notation

$$\Sigma + \chi : (\overline{\gamma_n})\gamma$$

just means that a declaration $\chi : (\overline{\gamma_n})\gamma$ is added to $\Sigma$.

In Fig. 8.6 the following conventions are used. $\overline{\alpha_{\gamma_n}}$ denotes the list $\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_n}$, with the understanding that the $\alpha_{\gamma_i}$ are distinct type variables. The first four rules in the type inference system in Fig. 8.6 are almost identical to the rules of Damas and Milner for ML typing [Dama82]. There are two differences: all inferences depend on the signature $\Sigma$ of the type algebra as well as the set of type assumptions $A$. Furthermore, generic instantiation in rule TAUT must respect $\Sigma$. This is written $\sigma \succeq_\Sigma \tau$ meaning that $\sigma$ has the form $\forall\overline{\alpha_{\gamma_n}}.\tau_0$, there are $\tau_i$ of sort $\gamma_i$ and $\tau = \tau_0[\tau_1/\alpha_{\gamma_1}, \ldots, \tau_n/\alpha_{\gamma_n}]$. The notation $FV(\tau)$ denotes the set of free type variables in $\tau$; $FV(\tau, A)$ denotes $FV(\tau) - FV(A)$.

If no class and instance declarations are present, every type constructor has the topmost sort as arity.

For a detailed discussion of the rules we refer to [Nipk91]. Notice that rule CLASS has no premises. The symbol ":" has two different meanings. On the one hand it assigns a type to an expression or a program. On the other hand it assigns a pair consisting of a typing hypothesis and a signature to a **class**- or **inst**-declaration.

already Schönfinkel has used it in [Scho24].

[12] Their "names" are bound variables!

[13] If $x \in \mathrm{dom}(A)$ its value will be "overwritten".

$$\text{TAUT}\quad \frac{A(x) \succeq_\Sigma \tau}{(A,\Sigma) \vdash x : \tau}$$

$$\text{APP}\quad \frac{(A,\Sigma) \vdash e_0 : \tau \longrightarrow \tau' \quad (A,\Sigma) \vdash e_1}{(A,\Sigma) \vdash \tau'}$$

$$\text{ABS}\quad \frac{(A + [x \mapsto \tau],\Sigma) \vdash e : \tau}{(A,\Sigma) \vdash \lambda x.e : \tau \longrightarrow \tau}$$

$$\text{LET}\quad \frac{(A,\Sigma) \vdash e_0 : \tau \; FV(\tau, A) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_k}\} \; (A + [x \mapsto \forall\overline{\alpha_{\gamma_k}}.\tau],\Sigma) \vdash e_1 : \tau'}{(A,\Sigma) \vdash \textbf{let}\ x = e_0\ \textbf{in}\ e_1 : \tau}$$

$$\text{CLASS}\quad (A,\Sigma) \vdash \textbf{class}\ \gamma \leq \gamma_1, \ldots, \gamma_n\ \textbf{where}\ x_1 : \forall\alpha_\gamma.\tau_1, \ldots, x_k : \forall\alpha_\gamma.\tau_k : (A + [x \mapsto \forall\alpha_\gamma.\tau_i \mid i = 1..k], \Sigma + \{\gamma \leq \gamma_j \mid j = 1..n\})$$

$$\text{INST}\quad \frac{A(x_i) = \forall\alpha_\gamma.\tau_i \qquad (A,\Sigma) \vdash e_i : \tau_i[\chi(\overline{\alpha_{\gamma_n}})/\alpha_\gamma] \qquad i = 1..k}{(A,\Sigma) \vdash \textbf{inst}\ \chi : (\overline{\gamma_n})\gamma\ \textbf{where}\ x_1 = e_1, \ldots, x_k = e_k : (A, \Sigma + \chi : (\overline{\gamma_n})\gamma)}$$

$$\text{PROG}\quad \frac{(A_{i-1},\Sigma_{i-1}) \vdash d_i : (A_i,\Sigma_i) \quad i = 1..n \qquad (A_n,\Sigma_n) \vdash e : \tau}{(A_0,\Sigma_0) \vdash d_1; \ldots : d_n; e : \tau}$$

Figure 8.6: The type inference rules for Mini-Haskell of Nipkow & Snelting

We have presented the simpler form of the type inference system as can be found in [Nipk91]. A problem is that the obtained order-sorted signature $\Sigma$ need not be regular. However, if we allow the formation of the conjunction of two sorts —- which corresponds to the `join` of two categories in Axiom — then the signature can be made regular (and downward complete). So we can assume w. l. o. g. that the signature is regular, omitting for simplicity the slightly more complicated type inference rules for the system handling these conjunctions of sorts. For more details we refer to [Nipk91].

The main result of [Nipk91] can be stated in the following form.

**Theorem 4. (Nipkow and Snelting)** The type inference problem for Mini-Haskell can be effectively reduced to the computation of order-sorted unifiers for a regular signature. It is thus decidable and there is a finite set of principal typings. If the signature is unitary unifying, then there is a unique principal type.

### Types of Functions

In this section we want to show that the above results on the type system for Haskell would allow an extension of the type system of Axiom.

In Axiom it is possible to have functions as objects, see [Jenk92, Sec. 6] and Fig. 8.7. Although Axiom has the concept of functions as objects and it can usually infer the type of objects, it cannot infer the type of functions.

Strictly speaking the inferred types `Void` or `FunctionCalled mersenne` in Fig. 8.7 are false, since they differ from the types when the functions are explicitly typed by the user.

The problem seems to be that Axiom can only infer ground types and not polymorphic types. For most purposes in computer algebra this might be sufficient. However, the type of

```
->fac n == if n < 3 then n else n * fac(n-1)
                                                              Type: Void
->fac 10
   (2)  3628800
                                                 Type: PositiveInteger
->g x == x + 1
                                                              Type: Void
->g 9
   Compiling function g with type PositiveInteger -> PositiveInteger
   (7)  10

                                                  Type: PositiveInteger
->g (2/3)
        5
   (8)  -
        3

                                                 Type: Fraction Integer

->mersenne i== 2**i - 1
                                                               Type: Void
->mersenne

                       i
   (2)  mersenne i == 2  - 1
                                           Type: FunctionCalled mersenne
->mersenne 3
   Compiling function mersenne with type PositiveInteger -> Integer

   (3)  7
                                                  Type: PositiveInteger
->addx  x == ((y :Integer): Integer +-> x + y)
                                                              Type: Void
>g:=addx 10
   Compiling function addx with type PositiveInteger -> (Integer ->
   Integer)

   (10)  theMap(*1;anonymousFunction;0;G1048;internal,502)
                                          Type: (Integer -> Integer)
```

Figure 8.7: Typing of some user-defined functions in Axiom

functions has to be polymorphic in many cases.

In Fig. 8.8 it is shown that Haskell can infer a type for such functions. The Haskell syntax has to be read as follows: `Integral` is a type class to which `Integer` belongs. The typing expression for `fact` has to be read as the type of `fact` is a function in one argument taking arguments of a type in type class `Integral` and returning an argument of the same type; the type variable `m` is bound in the expression and is chosen arbitrarily.

By Theorem 4 we know that it is decidable whether there is a typing of an expression and that there are only finitely many most principal typings in the positive case. As is discussed

```
fact 0 = 1
fact (n+1) = (n+1)*fact n
Phase TYPE:
fact :: Integral m => m -> m

square x = x * x
Phase TYPE:
square :: Num t => t -> t

mersenne i = 2^ i - 1
addx x    =  \y -> y+x
z::Integer
z=10
g = addx  z
h = g 3

Phase TYPE:
mersenne :: (Num tv57, Integral tv58) => tv58 -> tv57
addx :: Num tv59 => tv59 -> tv59 -> tv59
z :: Integer
g :: Integer -> Integer
h :: Integer
```

Figure 8.8: Corresponding typings in Haskell

in [Nipk91] the restrictions on typings in Haskell even imply that there is always a single principal type. However, since we do not know to what extend these assumptions will be justified in the area of computer algebra, we will not claim the more special result stated in Theorem 4.

For the purpose of this thesis we can stop at this point, since we are interested in questions of typability and not in ones of code generation. A certain problem in Haskell is that of *ambiguity*. Although all valid typings of an expression are instances of a most general type (involving type variables) it may happen that there is not enough information to generate code in an unambiguous way. Some discussions and examples of ambiguity can be found e. g. in [Huda99], [Faxe02], [Nipk91], However, since this problem arises "below" the typing level, some new concepts seem to be necessary in order to treat this problem formally, and the author of this thesis does not know of any such formal approaches.

### A Possible Application of Combining Type Classes and Parametric Polymorphism

As we have seen, we can extend a type system supporting type classes with parametric polymorphism and functions as first-class citizens and the type inference problem still remains decidable.

Such an extension of an Axiom like type system seems to be interesting in the area of computer algebra for several reasons. First of all lists play an important role in computer algebra and many typing issues related to lists are connected with parametric polymorphism.

But it seems to be possible to have some much further applications. As is shown by Rydeheard and Burstall in [Ryde88] it is possible to encode many concepts of category theory as types in ML and to state several constructive properties of category theory as ML programs. This encoding uses heavily the concepts of parametric polymorphism and higher-order functions. This formalism seems to be very useful, although there is no perfect correspondence between the objects of category theory and the types in ML.[14]

Now there are many well-known interactions between category theoretic concepts and algebraic concepts, see e. g. [Macl92, Sec. II.7] or [Mane76] for interactions of equational reasoning and category theory. Since many concepts in category theory are constructive, it seems to be possible to use some of these connections in a computer algebra system.

#### Typing of "Declared Only" Objects

Consider the Axiom dialogue:

```
->a:Integer
                                                    Type: Void

->a+a a is declared as being in Integer but has not been given a value.
```

Although a corresponding construct leads to a program error in Haskell, it could be typed by the Haskell type inference algorithm, if a declaration such as `a: Integer` would just add the corresponding typing assumption to the set of typing hypothesis.

Thus if we add a type declaration statement to the syntax of Mini-Haskell[15]

$$x \textbf{ has\_type } \tau,$$

then we simply need to add the following trivial rule to the ones given in Fig. 8.6:

$$\text{(TYPE-AS)} \qquad (A, \Sigma) \vdash x \textbf{ has\_type } \tau : (A + [x \mapsto \tau], \Sigma)$$

### 8.2.3 Complexity of Type Inference

#### The ML-fragment

The type inference problem for the simply typed lambda calculus, i. e. the ML core language without usage of `let` constructions reduces in linear time to a (syntactic) unification problem. Using a representation of terms as directed acyclic graphs (dags) the unification problem is decidable in linear time [Pate78], and so is the type inference problem.

In [Kane90, p. 450] this result is stated in the following precise form:

> Given a `let`-free expression $M$ of length $n$ (with all bound variables distinct), there is a linear time algorithm which computes a dag representation of the principal typing of $M$, if it exists, and returns *untypeable* otherwise. If it exists, the principal typing of $M$ has length at most $2^{O(n)}$ and dag size $O(n)$.

Even if `let`-expressions are used, the type inference problem remains decidable and can be solved using the Damas-Milner algorithm [Dama82]. Unfortunately, the complexity becomes

[14] For instance, the well-formedness of composites in a category is not a matter of type-checking, cf. [Ryde88, p. 58]. Other examples can be found in [Ryde88, Sec. 10].

[15] We will use **has_type** as an infix operation in the object language for the typing declaration instead of ":" in order to distinguish between the object and the meta level in rule (TYPE-AS).

dramatically worse. In the worst case, doubly-exponential time is required to produce a string output of a typing. Using a dag representation the algorithm can be modified to run in exponential time, which is also the proven lower (time complexity) bound of the problem (see e. g. [Kane90]).

Nevertheless, ML typing appears to be efficient in practice, although `let` expressions are frequently used in actual ML programs.[16]

#### Complexity of Type Inference for the System of Nipkow and Snelting

If no `let` expressions are used, then the type inference problem for the system of Nipkow and Snelting can be reduced to an unification problem for order-sorted terms.

This reduction is linear, so the inherent complexity of the problem is the same as the one of corresponding unification problem.

However, the resulting signature need not be regular. By introducing "conjunctive sorts" Nipkow and Snelting show how the signature can be made regular. This process consists of building new sorts for any finite subset of the set of sorts introduced by the `class` and `inst` declaration. This construction is thus exponential in the number of `class` and `inst` declaration of the program.

The unification problem for regular order-sorted signatures is decidable. However, in finite and regular signatures, deciding whether an equation is unifiable is an NP-complete problem (see [Smol89, Corollary 10]).

The situations is much better, if the signature is also coregular and downward complete, since in this case unification has quasi-linear complexity [Smol89, Theorem 18].

Since for many programs of the system the `class` and `inst` declarations are the same, the type inference problem is of feasible complexity if the obtained signature is coregular[17] and we view this signature as pre-computed.

Of course, if `let` statements are used, a lower bound bound for the complexity is exponentially. The complexity of various type systems for Haskell-like overloading has been investigated in [Volp91].

### 8.2.4 Algebraic Specifications of Type Classes

Many important classes of objects occurring in computer algebra can be defined by a finite set of equations, e. g. monoids, groups, Abelian groups, or rings.

So the corresponding type class can be specified by an algebraic specification (see e. g. [Ehri85], [Wirs91]) if we use the class of all models of the specification as the semantics of the specification, which is usually called the *loose semantics*.

**Remark** Usually, an algebraic specification is thought to specify abstract data types in the sense Axiom or Haskell. So very often the *initial semantics* is used, i. e. the specified object is the initial object in the category[18] of structures being models of the specification. A major advantage of this view is that many structures one is interested in — e. g. the rational numbers, stacks, queues, ... — can be specified by (sorted or order-sorted) equations. A

[16] We refer to [Kane90] for further discussions of this point.
[17] By construction, it is regular and downward complete.
[18] Category in the category theoretic sense!

characterization of structures which can be specified by the initial semantics can be found in [Hodg95].

So much of the work on algebraic specifications using the loose semantics are relevant for the specification type classes. Many references to such work are given in the survey of Wirsing [Wirs91].

### Some Hard-to-Specify Structures

Unfortunately, some very basic structures, namely integral domains (and fields) cannot be specified by equations, even if we allow equational implications. This is a consequence of the following simple fact.

**Lemma 4.** *The class of integral domains is not closed under the formation of products.*

**Proof** Let $A$, $B$ be two arbitrary integral domains (of cardinality $\geq 2$). Let $0 \neq a \in A$ and $0 \neq b \in B$. Then $(a,0) \cdot (0,b) = (0,0) = 0_{A \times B}$, i. e. the product $A \times B$ has zero divisors. □

The following well known theorem shows the problem.

**Theorem 5.** *A class $V$ of algebras*[19] *is definable by equational implications iff $V$ is closed under the formation of isomorphic images, products, subalgebras, and direct limits.*

**Proof** See [Grae79, p. 379]. □

Combining these results we obtain our claim.

**Corollary 5A** *The class of integral domains is not definable by equational implications.*

Since the technique of conditional term rewriting systems handles reasoning for equational implications (cf. [Klop90, Sec. 11], [Ders89]) even this powerful technique is to weak to be used as a mechanical tool for the specification of these examples.[20]

Clearly, integral domains or fields can be defined by a finite set of first-order formulas. Unfortunately, it is not possible to define them by Horn clauses, which would be one of the next classes of more powerful specification formalisms which are well known (cf. [Wirs91]) and have a much better computational behavior than arbitrary first-order formulas.[21]

**Proposition 1.** *Let $\mathcal{M}$ be a model-class of a first-order theory. If $\mathcal{M}$ is not closed under products, then the first-order theory of $\mathcal{M}$ cannot be axiomatized by a set of Horn sentences.*

**Proof** The claim follows immediately from the fact that Horn sentences are preserved under direct products (see e. g. [Chan90, Prop. 6.2.2]). □

Though most of the examples given as the "Basic Algebra Hierarchy" in [Jenk92] can be seen as model classes of finite sets of first-order sentences, there are some which are model classes of a set of first-order sentences — even if we allow infinite sets. An example is the category Finite.

**Lemma 5.** *There is no set of first-order sentences whose model class is the class of all finite sets.*

**Proof** If a set of first-order sentences has finite models of arbitrary large finite cardinality, then it also has an infinite model. □

[19] Algebra in the sense of universal algebra.
[20] At least, if we do not allow some coding of information.
[21] The success of PROLOG as a programming language is partly due to this fact.

**Remark** In [Dave90] it is shown that there are several quite simple operations in basic classes (such as integral domains) which cannot be defined constructively although they can be easily specified. So the meaning of a certain type class given there is that of a collection of all domains in which all the specified operations can be interpreted constructively. In [Dave91] the technique of introducing classes in which a operation can be defined constructively is applied to the problem of factorization of polynomials.

### Algebraic Theories

So it seems to be a wise decision in the design of Axiom to distinguish between "axioms" which are only stated in comments and give the intended meaning of an Axiom category as a class of algebraic structures and "attributes" that can be "explicitly expressed" [Jenk92, p. 522].

The parts which can be explicitly expressed by the Axiom system consists of equational properties only and are even a small subset of them. Applying the rich machinery of algebraic specifications techniques seems to be a possibility to extend the properties that are "explicitly expressed" considerably.

Moreover, there are many well known specifications of structures which are present as domains in Axiom. It seems to be an interesting field of further research to clarify the interaction between algebraically specified categories and algebraically specified domains.

The following extension of the work of Rector [Rect89] is a first approach in this direction: Assume that only finitely many sorts and operation symbols are used for the specification $\mathcal{D}$ of a certain domain and of the specification $\mathcal{C}$ of a certain type class. We can use different semantics as the initial semantics for the specification of the domain and the loose semantics for the specification of the type class. Then it can be deduced automatically whether the domain is a member of the type class in the following way: Generate the finitely many mappings which are potentially a view of $\mathcal{D}$ as $\mathcal{C}$ and check algorithmically whether this mapping is a view.[22] The possibility of giving certain specifications an initial semantics and of giving others a loose semantics is also built in OBJ (cf. [Wirs91], [Gogu92]). The former are called *objects*, the latter *theories* and there is the possibility to define certain mappings as views quite in the sense of above. However, the definition of views has "documentation aspect". A verification that a given mapping is a view is not implemented (cf. [Gogu92, Sec. 4.3]).

As we have seen it is not possible to specify all structures used in a computer algebra system by equations. Their are several possibilities to overcome this problem:

1. Use more powerful specification techniques.
2. Do not specify all structures *ab initio*, but take some of the structures as given.

The first possibility is used in [Limo92]. There the framework of first-order logic was chosen for the specification of structures arising in computer algebra. However, as we have shortly discussed, even this framework cannot handle all interesting cases.

Moreover, for an efficient system it is necessary that certain parts of a system have to be implemented by algorithms which are not the result of a formal specification. So the combination of taking certain parts as given and using equational reasoning for the formal part whose computational behavior is much better than the one of more powerful techniques seems to be a promising compromise between two contradicting requirements.

[22] We refer to [Rect89, p. 303] for the precise definitions of the used terms.

Another advantage of this approach is that already much is known about mathematical structures which can be specified in this way as e. g. the book by Manes on "Algebraic Theories" [Mane76] shows:

> The program of this book is to define for a "base category" $\mathcal{K}$ — a system of mathematical discourse consisting of objects whose structure we "take for granted" — categories of $\mathcal{K}$-objects with "additional structure," to prove general theorems about such algebraic[23] situations, and to present examples and applications of the resulting theory in diverse areas of mathematics.

#### Type Classes with Higher-Order Functions

Type inference remains decidable for a system with type classes even if higher-order functions are allowed in the way they are in Haskell. As we have shown in Sec. 8.2.2 such a combination is interesting for computer algebra systems.

In order to specify such a system algebraically it is necessary to extend the concepts of first-order algebraic specifications techniques with higher-order constructs. Some investigations of such combinations are done in [Brea89a] and in [Joua91]. The results given there show that such a combination has feasible properties, e. g. confluence and termination properties of the first-order part are preserved when some reasonable conditions are fulfilled.

### 8.2.5 Parameterized Type Classes

In Axiom categories can be parameterized. The occurring examples can be distinguished in several ways. On the one hand there is the distinction between domains and elements as parameters. On the other hand there are several other distinctions based on more "semantical" considerations.

Some parameterized type classes simply arise because the classes of algebraic objects should be described as being parameterized, e. g. vector spaces over a field $K$, or more generally, left- or right-modules over a ring $R$.

An example of a category having an element as a parameter is

```
PAdicIntegerCategory(p): Category == Definition where
  ++ This is the category of stream-based representations of
  ++ the p-adic integers.
```

It describes all domains implementing the $p$-adic integers for a given integer $p$.

This is an example of a class of categories used quite frequently in Axiom. The mathematical structures corresponding to the domains which belong to the category `PAdicIntegerCategory(p)` are all isomorphic! The reason for introducing such a category seems to be the following. For different computations it is useful to have different representations of the $p$-adic integers in a system.

The occurrence of categories in which all members are isomorphic (seen as mathematical structures) are not limited to categories having elements as parameters at all. Examples of others are

[23] Here "algebraic" means equationally definable.

```
UnivariatePolynomialCategory(R: Ring)
QuotientFieldCategory(D: IntegralDomain)
UnivariateTaylorSeriesCategory(Coef)
UnivariateLaurentSeriesCategory(Coef)
SquareMatrixCategory(ndim,R,Row,Col)
```

However, the case of elements as parameters for categories — which is claimed to be rare in [Jenk92, p. 524] — seems to be restricted to such categories.[24]

It seems to be useful to treat this class of type classes by a new concept and not only as a special case of the general one of type classes. The reason is the following: Formally, these type classes correspond exactly to the concept of abstract data type in the sense of algebraic specification as is e. g. defined by Wirsing [Wirs91]. Since the initial and the loose semantics coincide[25] the distinction between first-order and second-order types becomes a problem. However, such a distinction is very desirable, as we will show below.

### Sequences

In Axiom the operator `map` is defined by a simple overloading for several cases, such as matrices, vectors, quotient fields, ...

Using a parameterized type constructor `sequence` as in [Chen92] this form of ad-hoc polymorphism in Axiom could be changed to a form of type-class polymorphism. A parameterized category such as `HomogeneousAggregate` of the "data structure hierarchy" of Axiom seems to have almost the same intended meaning as `sequence`. So it seems to be possible even in Axiom to define `map` in `HomogeneousAggregate` and to have the algebraic examples as instances. In Sec. 8.3.2 we will use this view in order to show that many coercions will fulfill a condition that leads to a coherent type system.

### Type Inference

In [Chen92] an extension of the type system of Haskell is given allowing *types* as arguments in type classes. It is then proved that the type inference problem for parameterized type classes is decidable.

As we have argued above a restriction of category constructors to have domains as parameters only in Axiom does not seem to be a severe restriction for the type system of Axiom. In Sec. 8.4.2 we will show that not only type inference but even type checking for a system having types depending on elements is undecidable. The proof of undecidability given there can be easily applied to the case of categories having elements as parameters. So it seems to be useful not to allow elements as parameters for category constructors.

A certain problem in the proof given in [Chen92] is that an entirely new technique is used which cannot be seen as an extension of the approach of Nipkow and Snelting using order-sorted unification. However, such an extension would be desirable. Since we have to add other typing constructs to the language, it is desirable to have a well understood theory behind one aspect of the typing problem instead of using ad-hoc approaches.

[24] This was the result of an incomplete check of the source code of Axiom by the author.

[25] We will assume that there are only at most countable structures as members of a certain class. Most properties we are interested in are still valid if we look at the subclasses of classes which consist of at most countable structures, cf. [Hodg95].

Smolka [Smol88], [Smol89a] extends the framework of order-sorted algebras by introducing functions having sorts as parameters. So if we were looking at category constructors which take categories as arguments we could directly apply the results of Smolka. However, it is not clear whether these results are also useful for the cases we are interested in.

#### Algebraic Specifications of Parameterized Type Classes

As in the case of type classes, any specifications using the loose approach can be seen as specifications of parameterized type classes. In the survey of Wirsing [Wirs91] the relevant literature is cited. Especially, in [Wirs82] the important *pushout construction* for parameterized specifications has been studied.

### 8.2.6 Type Classes as First-Order Types

Categories in the type system of Axiom resp. type classes in the one of Haskell are second-order types.

By our general assumption first-order types have to correspond to structures in the sense of model theory or universal algebra.

We will briefly discuss to what extend this assumption is justified in various areas.

#### Group Theory

As the Axiom library shows the assumption of types corresponding to mathematical structures makes good sense for many objects of computer algebra with the exception of group theory programs. In a group theory program many algorithms take certain groups as input and return other groups — very often subgroups — as output. So it is reasonable to have the groups an algorithm works on as objects and not as types in a program. In this cases it seems to be more natural to treat certain classes of groups, such as the finitely presented groups, as a type, and not the groups themselves. Many of the algorithms of group theory depend on such a view of groups as objects. In this way groups are implemented in the group theory program GAP [GAPx17].

Some group theoretical functions can be found in general purpose computer algebra programs such as MAPLE (see e. g. [Char91a, Sec. 4.2]) or Axiom (see e. g. [Jenk92, App. E]). However, these are rather limited in power and coverage compared to the special group theory programs which have been developed in the last years (Cayley [Butl90], GAP [GAPx17]).

The observation above shows that it is difficult to come up with a design which can really integrate group theoretical algorithms and the ones of other areas of computer algebra. This problem can even be seen within Axiom. For instance, there are domains of permutation groups defined in Axiom. However, these domains are not members of the Axiom category group!

On the other hand it would be very desirable if some results of such group theoretic computations can be seen as types for other computations — such as the group of integers $\langle \mathbb{Z}, + \rangle$ or the finite cyclic groups $\langle \mathbb{Z}_m, + \rangle$.

Of course, if types become objects, then second-order types become first-order types. Nevertheless, the problem which has to be solved is that of the relationship between objects and

types, and not that of the relationship between types and type classes![26]

### Requirements of a System

If types are structures, then the type classes correspond to model classes of certain theories. Can we assume that such model classes do not appear as objects we will deal with?

Of course, as we have shown it makes good sense to view a type class as an algebraic object, namely the free term-algebra of order-sorted terms of the sort of the type class.

However, even if we model those order-sorted algebras within our system there is no need to view type classes as first-order types, as long as we use "isomorphic copies" of them. So we can even write e. g. a compiler or a type inference algorithm in our system using functions defined for those algebras.

The only thing we cannot model type safe are "run-time" interactions between such a compiler and an algebraic algorithm. But having systems which use self-modifying code is anyway contradicting the software-engineering principles we want to support by a type system.

As we have shown in Sec. 8.2.5 there are several type classes whose members are all isomorphic. For reasons of efficiency it is certainly necessary to distinguish these different members and to provide different type constructors for them, such as having a type constructor for univariate polynomials in sparse representation and another one for univariate polynomials in dense representation.

However, it might be useful on the level of a user interface to have only a *type constructor* "univariate polynomial" available for the user without forcing him to choose a particular representation.[27] In this case a *category constructor* univariate polynomial would become a *type constructor* inducing that certain type classes become first-order types.

Nevertheless, this seems to be useful only on the level of a user interface and seems to be restricted to cases in which the isomorphism between the types can be implemented in the system. Since such categories can be seen as (finite) equivalence classes in the coercion preorder (cf. Sec. 8.3.3), these equivalence classes could be easily implemented by a new special concept. Then there would still be a clear distinction between first-order types (which would include the constructs describing the equivalence classes) and the second-order types of type classes.

### Universal Algebra

In universal algebra, there are constructions which would imply the view of type classes as first-order objects. Namely, as in [Monk76, Sec. 24], one can construct for a class $\mathbf{K}$ of algebras the class $\mathbf{S}\,\mathbf{K}$ of substructures, or the class $\mathbf{P}\,\mathbf{K}$ of products or the class $\mathbf{H}\,\mathbf{K}$ of homomorphic images of $\mathbf{K}$.[28] Then many theorems can be stated as an equation, e. g. Birkhoff's theorem has the form

$$\mathbf{K} \text{ is a variety iff } \mathbf{K} = \mathbf{HSP}\,\mathbf{K}.$$

[26] See Chapter 8.4.2 of this thesis for further discussions.

[27] Contrary to a person implementing algorithms a user may be uncertain about the advantages of a particular representation so that the choice be the system might be better than the one of the user.

[28] More precisely, the class of structures which are *isomorphic* to substructures (or products, or homomorphic images) of elements of $\mathbf{K}$.

Although such a formulation is certainly elegant, it does not seem to be really necessary. So the additional difficulties which arise if one has to allow that type classes are members of the "equality type class" do not seem to be justified by the practical importance of such a construction.

In model theory the possibility of imposing an algebraic structure — e. g. the Lindenbaum algebra — or a topological structure on sets of formulas is used frequently. Via the correspondence between sets of formulas and model classes such a structure can also be imposed on a model class making it to an algebra or a topological space. However, since the properties on the side of the set of formulas are more useful people work with them and not with the model classes. Many books on model theory can serve as references for these remarks, some comprehensive ones are [Chan90], [Poiz85].

### Category Theory

The situation is different for category theory. An important tool for category theory is the possibility to have a category of all (small)[29] categories as objects and the functors as arrows, or having functor categories, etc.

In this case it is not possible to have a perfect correspondence between types and type classes in our system and the objects of category theory. More generally, it is not possible to have such a perfect correspondence between the concepts of category theory and a *predicative*[30] type-theory such as Martin-Löf's type theory [Mart80], as is also discussed in [Ryde88, Sec. 10]. This is certainly a problem since impredicative type theories might have unwanted properties. Impredicative variants of Martin-Löf's system can have an undesirable computational behavior, as is discussed e. g. in [Meye86], [Howe87], [Coqu86].[31]

So it might be preferable to have a type system which allows some modeling of category theory but not a perfect correspondence.

### Bounded Polymorphism

So in the main area of computer algebra there seems to be no need for a concept of type classes as first-order types. So we will only sketch some language proposals in which such a concept could be modeled. The main idea is to have first-order types as "bounds" to polymorphic constructs.

The notion of *bounded quantification* was introduced by Cardelli and Wegner [Card85] in the language Fun. This proposed language integrated Girard-Reynolds polymorphism [Gira72], [Reyn74] with Cardelli's first-order calculus of subtyping [Card88].

**Remark** The so called "second-order polymorphic $\lambda$-calculus" was rediscovered independently by Reynolds [Reyn74] as a formalism to express "polymorphism" in programming languages. Girard has introduced his system $F$ as a proof theoretic tool to give a consistency proof for second-order Peano arithmetic along a line of proof theoretic research which has originated with Gödel [Gode58]. A proof that all $\lambda$-terms typeable in system $F$ are

[29] Small means that the categories are sets in a set theory and not proper classes.

[30] The word "predicative" refers to the fact that a universe of types is introduced only after all of its members are introduced.

[31] This problem is discussed in the literature under the names *"Type: Type"* — referring to the problem whether the collection of all types is a type — or *Girard's Paradox*, since Girard has shown in his thesis [Gira72] that the original version of Martin-Löf's type theory allowing such constructs is inconsistent with intuitionistic mathematics which it was supposed to model.

strongly normalizable and that this theorem implies the consistency of second-order Peano arithmetic can be found in the book by Girard, et al. [Gira89].

Fun and its relatives have been studied extensively by programming language theorists and designers. A slight modification of this language — called *minimal Bounded Fun* or $F_{\leq}$ — by Curien and Ghelli was extensively studied by Pierce in his thesis [Pier91]. Unfortunately, the type checking problem for this language was proven to be undecidable by Pierce [Pier91], [Pier91a].

Syntactically, types can have the form

$$\forall \alpha \leq \sigma_1 \,.\, \sigma_2,$$

where $\alpha$ is a type variable and $\sigma_1$ and $\sigma_2$ are types. Besides the usual rules asserting reflexivity and transitivity of $\leq$ the following rule is essential:[32]

$$\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma, \alpha \leq \tau_1 \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \forall \alpha \leq \sigma_1 \,.\, \sigma_2 \;\leq\; \forall \alpha \leq \tau_1 \,.\, \tau_2} \qquad \text{(Sub-All)}$$

The expressiveness of the language[33] comes from the fact that first-order types are bounds for type variables. The rule

$$x \in V_{\sigma'} \text{ and } \sigma' \leq \sigma \implies x \in T_{\Sigma}(V)_{\sigma}$$

constituting a part of the definition of order-sorted terms (cf. Def. 8) can be seen as a special form of rule (Sub-All) if one would restrict the system $F_{\leq}$ to cases which distinguish between two kinds of types where only one kind is allowed to be a bound. The typing rules for Mini-Haskell (cf. Fig. 8.6) could be simulated by the typing rules for $F_{\leq}$ using a similar distinction between types.

We will not develop a formal interpretation of Mini-Haskell in $F_{\leq}$ which could be done along the lines sketched above because it is not clear yet whether the additional expressiveness of $F_{\leq}$ is useful for a computer algebra system or an extension by another system would be more appropriate.

**Relation to Object-Oriented Programming**

There has been a lot of work in the last years to show how the notions of *object-oriented programming*[34] can be modeled in a type safe way by using $F_{\leq}$ or a related system like the so called $F$-bounded polymorphic second-order lambda calculus [Cann89]. Some experimental languages based on such principles are TOOPL [Bruc93] and Quest [Card91].

As is argued e. g. in [Limo92], [Temp92] and can be seen by a language for symbolic computation as VIEWS [Abda86] the principles of object-oriented programming are important tools for the design of a computer algebra system.

However, as we have shown in Sec. 8.2.1 and is discussed in more detail in [Huda92], [Berg92] some important principles of object-oriented programming already come with the use of type classes.

There are some examples — e. g. ones related to problems of strict versus non-strict inheritance (see e. g. [Limo92],[Temp92] — which cannot be expressed in the type system of

[32] For a detailed discussion of the rules we refer to the thesis of Pierce [Pier91].
[33] Since type checking is undecidable, it might be too expressive.
[34] Some books on object-oriented programming and languages are [Meye88], [Gold83], [Kirk89], [Birt80], [Stro95].

Axiom and which could be expressed using more sophisticated techniques of object-oriented programming. However, as we will show in Sec. 8.3.3 there are properties of a type system which cannot be expressed by mechanisms of object-oriented programming alone but require an additional concept. So it may be preferable to use a system which is as simple as possible, even if not every example can be expressed in it.[35]

## 8.3 Coercions

In mathematics the convention to identify an object with its image under an embedding is used frequently. It is certainly one of sources of strength of mathematical notation. Very often certain structures are constructed as being of quite different shape and then this convention is used to identify one with a certain subset of another one. Some examples which are explained in many textbooks are the "subset relationship"

$$\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C},$$

embeddings of elements of $\mathbb{Q}$ in algebraic extensions of $\mathbb{Q}$ or in a $p$-adic completion, or the embeddings of elements of a commutative ring $R$ in $R[x]$, ...

If these mathematical structures correspond to types in a system and the embeddings are computable functions, then this convention can be modeled by the use of *coercions*.

While the use of implicit conversions instead of explicit conversions might be debatable for parts of a system in which new efficient algorithms have to be written, it is certainly necessary for a user interface.

### 8.3.1 General Remarks

We will assume that we have a mechanism to declare some functions between types to be *implicit coercions* between these types (or simply *coercions*). If there is a coercion $\phi : t_1 \longrightarrow t_2$ we will write $t_1 \trianglelefteq t_2$.

**Remark** The requirement of set theoretic ground types and coercion functions excludes some constructions — if we gave all types the "obvious" set theoretic interpretation —, as the one used in in [Mitc91, Lemma 2], which assumes a coercion from the space of functions $\texttt{FS}(D, D)$ over some domain $D$ into this domain. Such coercions which correspond to certain constructions of models of the $\lambda$-calculus (see e. g. [Bare84]) seem to be of theoretical interest only. At least for the purpose of a computer algebra system the requirement of set theoretic coercion functions does not seem to be a restriction at all!

### 8.3.2 Coherence

In a larger system, it is possible that there are different ways to have a coercion from one type into another. Following [Brea91] and [Reyn91] we will call a type system *coherent*, if the coercions are independent of the way they are deduced in the system.[36]

[35] There seems to be one single example which is used by several authors — e. g. in [Limo92] and in [Baum95] — implying the need of non-strict inheritance in a computer algebra system!

[36] Notice that the term "coherence" is used similarly in category theory (see e. g. [Macl91]) but is used quite differently in connection with order-sorted algebras (e. g. in [Wald92], [Gogu92], [Rect89]).

In the following we will look at different kinds of coercions which occur and we will state some conditions which will yield the coherence of the system. Besides the technical proof of the coherence theorem we will give some informal discussions about the significance of these conditions.

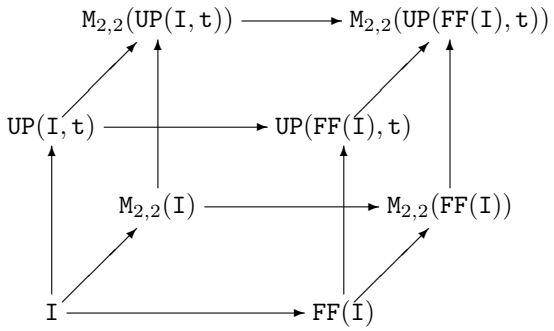
### Motivating Examples

Consider the expression

$$\mathtt{t} - \begin{pmatrix} 1 & 0 \\ 3 & \frac{1}{2} \end{pmatrix}$$

which — as a mathematician would conclude — denotes a $2 \times 2$-matrix over $\mathbb{Q}[\mathtt{t}]$ where $\mathtt{t}$ is the usual shorthand for $\mathtt{t}$ times the identity matrix. In an Axiom like type system, this expression involves the following types and type constructors: The integral domain `I` of integers, the unary type constructor `FF` which forms the quotient field of an integral domain, the binary type constructor `UP` which forms the ring of univariate polynomials over some ring in a specified indeterminate, and the type constructor $\mathtt{M}_{2,2}$ building the $2 \times 2$-matrices over a commutative ring.

In order to type this expression correctly several of the following coercions have to be used.




There are different ways to coerce `I` to $\mathtt{M}_{2,2}(\mathtt{UP}(\mathtt{FF}(\mathtt{I}), \mathtt{t}))$. Of course one wants the embedding of `I` in $\mathtt{M}_{2,2}(\mathtt{UP}(\mathtt{FF}(\mathtt{I}), \mathtt{t}))$ to be independent of the particular choice of the coercion functions.

In this example this independence seems to be the case, but how can we *prove* it? Moreover, not all coercions which would be desirable for a user share this property. Consider e. g. the binary type constructor "direct sum" $\oplus$ defined for Abelian groups. One could coerce $A$ into $A \oplus B$ via a coercion $\phi_1$ and $B$ into $A \oplus B$ via a coercion $\phi_2$. But then the image of $A$ in $A \oplus A$ depends on the choice of the coercion function!

### Definition

Relying on the set theoretic semantics for our types and coercion functions we can give the following definition of coherence.

**Definition 20. (Coherence)** *A type system is* coherent *if the following condition is satisfied:*

> *For any ground types $t_1$ and $t_2$ of the type system, if $\phi, \psi : t_1 \longrightarrow t_2$ are coercions then $\phi = \psi$.*

### General Assumptions

It will be convenient to declare each identity function on a type to be an implicit coercion.

**Assumption 1.** *For any ground type $t$ the identity on $t$ will be a coercion. If $\phi : t_1 \longrightarrow t_2$ and $\psi : t_2 \longrightarrow t_3$ are coercions, then the composition $\phi \circ \psi : t_1 \longrightarrow t_3$ of $\phi$ and $\psi$ is a coercion.*

**Lemma 6.** *If assumption 1 holds, then the set of ground types as objects together with the coercion functions as arrows form a category.*

**Proof** Since composition of functions is associative and the identity function is a coercion, all axioms of a category are fulfilled. □

In the following we will always assume that assumption 1 holds even if we do not mention it explicitly.

### Base Types

It is a good instrument for structuring data types to have only as few types as possible as base types but to construct them by a type constructor whenever possible.[37]

Since there are only very few coercions between base types the following assumption seems to be easily satisfiable.

**Assumption 2. (Base Types)** The subcategory of base types and coercions between base types forms a preorder, i. e. if $t_1$ and $t_2$ are base types and $\phi, \psi : t_1 \longrightarrow t_2$ are coercions then $\phi = \psi$.

### Structural Coercions

**Definition 21. (Structural Coercions)** *The $n$-ary type constructor ($n \geq 1$) $f$ induces a* structural coercion, *if there are sets $\mathcal{A}_f \subseteq \{1, \ldots, n\}$ and $\mathcal{M}_f \subseteq \{1, \ldots, n\}$ such that the following condition is satisfied:*

*Whenever there are declarations $f : (\sigma_1 \cdots \sigma_n)\sigma$ and $f : (\sigma'_1 \cdots \sigma'_n)\sigma'$ and ground types $t_1 : \sigma_1, \ldots, t_n : \sigma_n$ and $t'_1 : \sigma'_1, \ldots, t'_n : \sigma'_n$ such that $t_i = t'_i$ if $i \notin \mathcal{A}_f \cup \mathcal{M}_f$ and there are coercions*

$$\begin{array}{ll} \phi_i : t_i \longrightarrow t'_i, & \text{if } i \in \mathcal{M}_f, \\ \phi_i : t'_i \longrightarrow t_i, & \text{if } i \in \mathcal{A}_f, \\ \phi_i = \mathrm{id}_{t_i} = \mathrm{id}_{t'_i}, & \text{if } i \notin \mathcal{A}_f \cup \mathcal{M}_f, \end{array}$$

*then there is a* uniquely defined *coercion*

$$\mathcal{F}_f(t_1, \ldots, t_n, t'_1, \ldots, t'_n, \phi_1, \ldots, \phi_n) : f(t_1, \ldots, t_n) \longrightarrow f(t'_1, \ldots, t'_n).$$

*The type constructor $f$ is* covariant in its $i$-th argument, *if $i \in \mathcal{M}_f$. It is* contravariant in its $i$-th argument, *if $i \in \mathcal{A}_f$.*

Instead of the adjective "covariant" we will sometimes use the adjective "monotonic", and instead of "contravariant" we will sometimes use "antimonotonic", because both terminologies are used in the literature and reflect different intuitions which are useful in different contexts.

[37] As an example consider the field of rational numbers, which can be constructed as the quotient field of the integers.

**Assumption 3. (Structural Coercions)** *Let $f$ be $n$-ary type constructor which induces a structural coercion and let $f(t_1,\ldots,t_n)$, $f(t'_1,\ldots,t'_n)$, and $f(t''_1,\ldots,t''_n)$ be ground types. Assume that*

$$\begin{array}{ll} t_i \trianglelefteq t'_i \trianglelefteq t''_i, & \text{if } i \in \mathcal{M}_f, \\ t''_i \trianglelefteq t'_i \trianglelefteq t_i, & \text{if } i \in \mathcal{A}_f, \\ t_i = t'_i = t''_i, & \text{if } i \notin \mathcal{A}_f \cup \mathcal{M}_f. \end{array}$$

*and let $\phi_i : t_i \longrightarrow t'_i$, $\phi'_i : t'_i \longrightarrow t''_i$ (if $i \in \mathcal{M}_f$), and $\phi'_i : t''_i \longrightarrow t'_i$, $\phi_i : t'_i \longrightarrow t_i$ (if $i \in \mathcal{A}_f$) be coercion functions. For $i \notin \mathcal{A}_f \cup \mathcal{M}_f$ let $\phi_i$ and $\phi'_i$ be the appropriate identities.*

*Then the following conditions are satisfied:*

1. $\mathcal{F}_f(t_1,\ldots,t_n,t_1,\ldots,t_n,\mathrm{id}_{t_1},\ldots,\mathrm{id}_{t_n})$ *is the identity on* $f(t_1,\ldots,t_n)$,
2. $\mathcal{F}_f(t_1,\ldots,t_n,t''_1,\ldots,t''_n,\phi_1\circ\phi'_1,\ldots,\phi_n\circ\phi'_n) =$
   $\mathcal{F}_f(t_1,\ldots,t_n,t'_1,\ldots,t'_n,\phi_1,\ldots,\phi_n)\circ\mathcal{F}_f(t'_1,\ldots,t'_n,t''_1,\ldots,t''_n,\phi'_1,\ldots,\phi'_n)$.

Let $f : (\sigma_1\cdots\sigma_n)\sigma$ be an $n$-ary type constructor which induces a structural coercion. Let $\mathcal{C}_{\sigma_i}$ be the category of ground types of sort $\sigma_i$ as objects and the coercions as arrows, let $\mathcal{C}_{\sigma_i}^{\mathrm{op}}$ be the dual category of $\mathcal{C}_{\sigma_i}$ and let $\mathcal{C}_{\sigma_i}^{\mathrm{triv}}$ be the discrete subcategory of the objects of $\mathcal{C}_{\sigma_i}$. Define

$$\mathcal{C}_i = \begin{cases} \mathcal{C}_{\sigma_i}, & \text{if } i \in \mathcal{M}_f, \\ \mathcal{C}_{\sigma_i}^{\mathrm{op}}, & \text{if } i \in \mathcal{A}_f, \\ \mathcal{C}_{\sigma_i}^{\mathrm{triv}}, & \text{if } i \notin \mathcal{A}\cup\mathcal{M}_f. \end{cases}$$

Then assumption 3 means that the mapping assigning $f(t_1,\ldots,t_n)$ to the $n$-tuple $(t_1,\ldots,t_n)$ and assigning the coercion

$$\mathcal{F}_f(t_1,\ldots,t_n,t'_1,\ldots,t'_n,\phi_1,\ldots,\phi_n)$$

to the $n$-tuple $(\phi_1,\ldots,\phi_n)$ of coercions is a *functor* from

$$\mathcal{C}_1\times\cdots\times\mathcal{C}_n$$

into $\mathcal{C}_\sigma$.

Typical examples of type constructors which induce a structural coercion are `list`, `UP`, $\mathtt{M}_{n,n}$, `FF`. These examples give rise to structural coercions, because the constructed type can be seen as an instance of the parameterized type class `sequence` (cf. Sec. 8.2.5).[38] The coercions between the constructed types are then obtained by *mapping* the coercions between the type parameter into the sequence. Since a mapping of functions distributes with function composition, assumption 3 will be satisfied by these examples.

Although many examples of structural coercions satisfying assumption 3 can be explained by this mechanism, there are others, which will satisfy assumption 3 because of another reason, so that the more general framework we have chosen is justified. For instance, it is another mechanism which gives rise to the structural coercion in the case of the "function space" type constructor, as is well known.[39] It is contravariant in its first argument and covariant in its second argument, as the following considerations show: Let $A$ and $B$ be two types where there is an implicit coercion $\phi$ from $A$ to $B$. If $f$ is a function from $B$ into a type $C$, then $f\circ\phi$ is a function from $A$ into $C$. Thus any function from $B$ into $C$ can be coerced into a function from $A$ into $C$. Thus an implicit coercion from $\mathtt{FS}(B,C)$ into $\mathtt{FS}(A,C)$ can be defined, i. e. $\mathtt{FS}(B,C)\trianglelefteq\mathtt{FS}(A,C)$. If $C\trianglelefteq D$ by an implicit coercion $\psi$, then $\psi\circ f$ is a function

[38] The sequences can be of fixed finite length, as in the case `FF` where it consists of two elements only, the numerator and the denominator.

[39] See e. g. [Card86].

from $A$ into $D$, i. e. an implicit coercion from $\texttt{FS}(A, C)$ into $\texttt{FS}(A, D)$ can be defined. In this case assumption 3 is satisfied because of the associativity of function-composition.

Although many important type constructors arising in computer algebra are monotonic in all arguments it is not justified to assume that this property will always hold as was done in [Como91]. We have already seen that the type constructor for building "function spaces" is antimonotonic in its first argument. Constructions like the fixpoint field of a certain algebraic extension of $\mathbb{Q}$ under a group of automorphisms in Galois theory (see e. g. [Zari75], [Marc77], [Lang05]) would give other — more algebraic examples — of type constructors which are antimonotonic.[40]

However, an assumption that all type constructors are monotonic or antimonotonic in all arguments as in [Fuhx90], [Mitc91] still seems to be too restrictive for our purposes.

If one allows a type constructor building references (pointers) to objects of a certain type as is possible in Standard ML or in the system described by Kaes [Kaes92], then this type constructor is neither monotonic nor antimonotonic.

There are also algebraic examples of type constructors which are neither monotonic nor antimonotonic. Consider e. g. the quotient groups $G/G'$, where $G'$ is the derived subgroup of $G$ (see e. g. [Robi96, p. 28]). Assume that $H$ can be embedded in $G$. Then in general it is not possible to embed $H/H'$ in $G/G'$ or vice versa. Thus if one would have a type constructor building the type $G/G'$ for a given group $G$, then this type constructor would be neither monotonic nor antimonotonic.

**Remark** Of course, one has to restrict the groups in consideration to ones for which the construction of $G/G'$ can be performed effectively. One such class of groups is that of the finite polycyclic groups (cf. [GAPx17]).

### Direct Embeddings in Type Constructors

**Definition 22. (Direct Embeddings)** *Let $f : (\sigma_1, \ldots, \sigma_n)\sigma$ be a $n$-ary type constructor. If for some ground types $t_1 : \sigma_1, \ldots, t_n : \sigma_n$ there is a coercion function*

$$\Phi^i_{f,t_1,\ldots,t_n} : t_i \longrightarrow f(t_1, \ldots, t_n),$$

*then we say that $f$* has a direct embedding at its $i$-th position.

*Moreover, let*

$$\mathcal{D}_f = \{i \mid f \text{ has a direct embedding at its } i\text{-th position}\}$$

*be the* set of direct embedding positions of $f$.

**Remark** In Axiom the inverses of direct embeddings are called *retractions* (cf. [Jenk92, p. 713]) assuming that the direct embeddings are always injective. Thus the usage of the term in Axiom is a special case of our usage of that term, since in our terminology any partial function which is an inverse of any injective coercion can be a retraction.

On the other hand the Axiom terminology shows that the designers of Axiom have seen the importance of direct embeddings, even if there is no special terminology for direct embeddings themselves but only for their inverses!

[40] In GAP [GAPx17] such constructs are implemented as functions and not as type constructors, cf. the discussion in Sec. 8.2.6. Nevertheless, the implementation as type constructors seems to be a reasonably possibility.

**Remark** In a system, a type constructor represents a parameterized abstract data type which is usually built uniformly from its parameters. So the family of coercion functions

$$\{\Phi^i_{f,t_1,\ldots,t_n} \mid t_i \in T_\Sigma(\{\})_{\sigma_i}\}$$

will very often be just one (*polymorphic*) function. In this respect the situation is similar to the one in Sec. 8.3.2.

**Assumption 4. (Direct Embeddings)** *Let $f : (\sigma_1 \cdots \sigma_n)\sigma$ be a $n$-ary type constructor. Then the following conditions hold:*

1. $|\mathcal{D}_f| \leq 1$.
2. *The coercion functions which give rise to the direct embedding are unique, i. e. if $\Phi^i_{f,t_1,\ldots,t_n} : t_i \longrightarrow f(t_1,\ldots,t_n)$ and $\Psi^i_{f,t_1,\ldots,t_n} : t_i \longrightarrow f(t_1,\ldots,t_n)$, then*

$$\Phi^i_{f,t_1,\ldots,t_n} = \Psi^i_{f,t_1,\ldots,t_n}.$$

Many important type constructors such as `list`, $\mathtt{M}_{n,n}$, `FF`, and in general the ones describing a "closure" or a "completion" of a structure — such as the $p$-adic completions or an algebraic closure of a field — are unary. Since for unary type constructors the condition $|\mathcal{D}_f| \leq 1$ is trivial and the second condition in assumption 4 should be always fulfilled, the assumption holds in this cases.

For $n$-ary type constructors ($n \geq 2$) the requirement $|\mathcal{D}_f| \leq 1$ might restrict the possible coercions. Consider the "direct sum" type constructor for Abelian groups which we have already seen that it could lead to a type system that is not coherent if we do not restrict the possible coercions. For a type constructor

$$\oplus : (\mathsf{Abelian\_group}\ \mathsf{Abelian\_group})\mathsf{Abelian\_group}$$

the requirement $|\mathcal{D}_f| \leq 1$ means that it is only possible to have either an embedding at the first position or at the second position.

In the framework that we have used the types $A \oplus B$ and $B \oplus A$ will be different. However, the corresponding mathematical objects are *isomorphic*. Having a mechanism in a language that represents certain isomorphic mathematical objects by the same type (cf. Sec. 8.3.3) the declaration of both natural embeddings to be coercions would not lead to an incoherent type system. Notice that such an additional mechanism, which corresponds to factoring the free term-algebra of types we regard by some congruence relation, will be a conservative extension for a coherent type system. If a type system was coherent, it will remain coherent. It is only possible that a type system being incoherent otherwise becomes coherent.

Let $f : (\sigma\sigma')\sigma$ be a binary type constructor with $\sigma$ and $\sigma'$ incomparable having direct embeddings at the first and second position, and let $t : \sigma$ and $t' : \sigma'$ be ground types such that

$$t' \trianglelefteq f(f(t,t'),t').$$

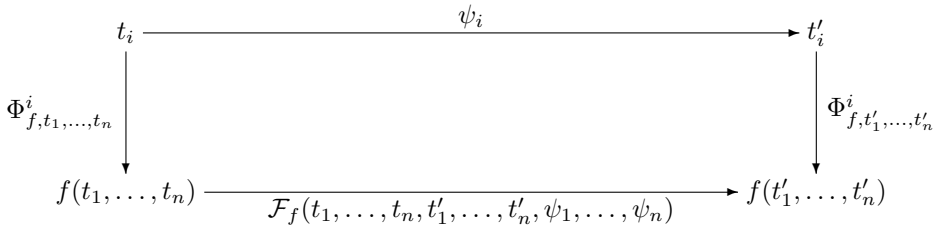
Then there are two possibilities to coerce $t'$ into $f(f(t,t'),t')$ which might be different in general. In the case of types $\mathtt{R} : \mathsf{c\_ring}$ and $\mathtt{x} : \mathsf{symbol}$ the coercions of $\mathtt{x}$ into $\mathtt{UP}(\mathtt{UP}(\mathtt{R},\mathtt{x}),\mathtt{x})$ are unambiguous, if $\mathtt{UP}(\mathtt{UP}(\mathtt{R},\mathtt{x}),\mathtt{x})$ and $\mathtt{UP}(\mathtt{R},\mathtt{x})$ are the same type. However, it does not seem to be generally possible to avoid the condition $|\mathcal{D}_f| \leq 1$ even in cases where a type constructor is defined for types belonging to incomparable type classes.

The naturally occurring direct embeddings for types built by the type constructors `FF` and `UP` show that in the context of computer algebra there are cases in which a coercion is defined

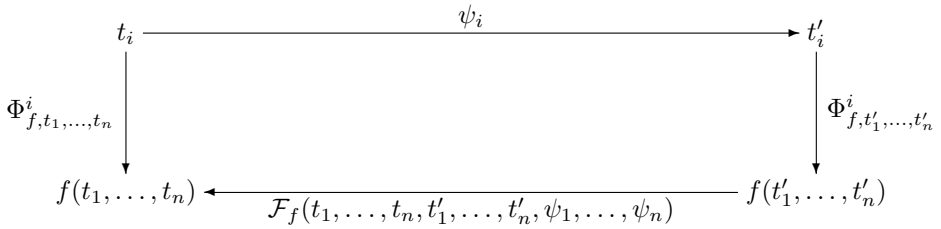
into a type belonging to an incomparable type class, into a type belonging to a more general type class, into a type belonging to a less general type class, or into a type belonging to the same type class. So coercions occur quite "orthogonal" to the inheritance hierarchy on the type classes showing an important difference between the coercions in computer algebra and the "subtypes" occurring in object oriented programming (cf. Sec. 8.3.3).

The next assumption will guarantee that structural coercions and direct embeddings will interchange nicely.

**Assumption 5. (Structural Coercions and Embeddings)** *Let $f$ be a $n$-ary type constructor which induces a structural coercion and has a direct embedding at its $i$-th position. Assume that $f : (\sigma_1 \cdots \sigma_n)\sigma$ and $f : (\sigma'_1 \cdots \sigma'_n)\sigma$, $t_1 : \sigma_1, \ldots, t_n : \sigma_n$, and $t'_1 : \sigma'_1, \ldots, t'_n : \sigma'_n$. If there are coercions $\psi_i : t_i \longrightarrow t'_i$, if the coercions $\Phi^i_{f,t_1,\ldots,t_n}$ and $\Phi^i_{f,t'_1,\ldots,t'_n}$ are defined, and if $f$ is covariant at its $i$-th argument, then the following diagram is commutative:*




*If $f$ is contravariant at its $i$-th argument, then the following diagram is commutative:*




The type constructors `list`, `UP`, $\mathtt{M}_{n,n}$ may serve as examples of constructors which induce structural coercions and can also have direct embeddings: It might be useful to have coercions from elements into one element lists, from elements of a ring into a constant polynomial or to identify a scalar with its multiple with the identity matrix.

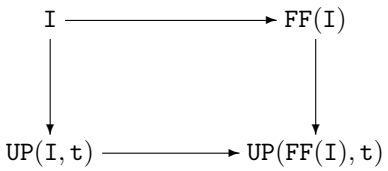
As was already discussed in Sec. 8.3.2, in all these examples the parameterized data types can be seen as sequences and the structural coercions — i. e. $\mathcal{F}_{\mathtt{UP}}(\mathtt{I}, \mathtt{x}, \mathtt{FF}(\mathtt{I}), \mathtt{x}, \psi, \mathrm{id}_{\mathtt{x}})$ — can be seen as a kind of "mapping" operators.

The direct embeddings are "inclusions" of elements in these sequences. Since applying a coercion function to such an elements and then "including" the result in a sequence will yield the same result as first including the element in the sequence and then "mapping" the coercion function into the sequence, assumption 5 will be satisfied by these examples. For instance,

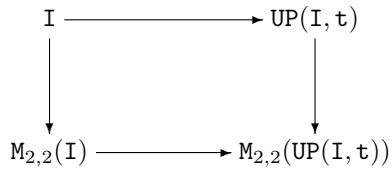
$$\mathcal{F}_{\mathtt{UP}}(\mathtt{I}, \mathtt{x}, \mathtt{FF}(\mathtt{I}), \mathtt{x}, \Phi^1_{\mathtt{FF},\mathtt{I}}, \mathrm{id}_{\mathtt{x}})$$

is the function which maps the coercion function $\Phi^1_{\mathtt{FF},\mathtt{I}}$ to the sequence of elements of `I` in $\mathtt{UP}(\mathtt{I}, \mathtt{x})$ which represents the polynomial.

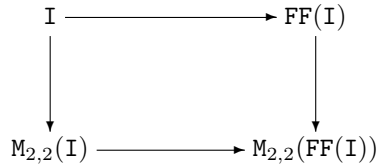
Thus the diagrams




and




and




which are instances of the diagrams in assumption 5 are commutative.[41]

If the mathematical structure represented by a type $t_i$ in assumption 5 has non-trivial automorphisms, then it is possible to construct the structural coercion

$$\mathcal{F}_f(t_1, \ldots, t_n, t'_1, \ldots, t'_n, \psi_1, \ldots, \psi_n)$$

in a way such that the assumption is violated: just apply a non-trivial automorphism to $t_i$! However, such a construction seems to be artificial. Moreover, the argument shows that a possible violation of assumption 5 "up to an automorphism" can be avoided by an appropriate definition of

$$\mathcal{F}_f(t_1, \ldots, t_n, t'_1, \ldots, t'_n, \psi_1, \ldots, \psi_n).$$

### A Coherence Theorem

We are now ready to state the main result of this section. The assumptions 1, 2, 3, 4, and 5 are "local" coherence conditions imposed on the coercions of the type system. In the following theorem we will prove that the type system is "globally" coherent, if these local conditions are satisfied.

**Theorem 6. (Coherence)** *Assume that all coercions between ground types are only built by one of the following mechanisms:*

1. *coercions between base types;*
2. *coercions induced by structural coercions;*
3. *direct embeddings in a type constructor;*
4. *composition of coercions;*
5. *identity function on ground types as coercions.*

[41] The first of these diagrams can also be found in [Fort90].

*If the assumptions 1, 2, 3, 4, and 5 are satisfied, then the set of ground types as objects and the coercions between them as arrows form a category which is a preorder.*

**Proof** By assumption 1 and lemma 6 the set of ground types as objects and the coercions between them as arrows form a category.

For any two ground types $t$ and $t'$ we will prove by induction on the complexity of $t'$ that if $\phi, \psi : t \longrightarrow t'$ are coercions then $\phi = \psi$ which will establish the theorem.

If $\mathrm{com}(t') = 1$ then we have $\mathrm{com}(t) = 1$ because of the assumption on the possible mechanisms for building coercions. Since $\mathrm{com}(t) = 1$ and $\mathrm{com}(t') = 1$ the claim follows from assumption 2.

Now assume that the induction hypothesis holds for $k$ and let $\mathrm{com}(t') = k + 1$. Thus we can assume that $t' = f(u_1, \ldots, u_n)$ for some $n$-ary type constructor $f$.

Let $\phi, \psi : t \longrightarrow t'$ be coercions.

The coercions $\phi$ and $\psi$ are compositions of coercions between base types, direct embeddings in type constructors and structural coercions. Because of assumption 3 and the induction hypothesis we can assume that there are ground types $s_1$ and $s_2$ and unique coercions $\psi_1 : t \longrightarrow s_1$ and $\psi_2 : t \longrightarrow s_2$ such that

$$\phi = \mathcal{F}_f(\ldots, t, \ldots, s_1, \ldots, \psi_1, \ldots) \tag{8.1}$$

or

$$\phi = \psi_1 \circ \Phi^i_{f,\ldots,s_1,\ldots} \tag{8.2}$$

Similarly,

$$\psi = \mathcal{F}_f(\ldots, t, \ldots, s_2, \ldots, \psi_2, \ldots) \tag{8.3}$$

or

$$\psi = \psi_2 \circ \Phi^j_{f,\ldots,s_2,\ldots} \tag{8.4}$$

If $\phi$ is of form 8.1 and $\psi$ is of form 8.3, then $\phi = \psi$ because of assumption 3 and the uniqueness of $\mathcal{F}_f$. If $\phi$ is of form 8.2 and $\psi$ is of form 8.3, then $\phi = \psi$ because of assumption 5. Analogously for $\phi$ of form 8.1 and $\psi$ of form 8.4.

If $\phi$ is of form 8.2 and $\psi$ is of form 8.3 then assumption 4 implies that $i = j$ and $s_1 = s_2$. Because of the induction hypothesis we have $\psi_1 = \psi_2$ and hence $\phi = \psi$ again by assumption 4. □

### 8.3.3 Type Isomorphisms

In several important cases there is not only a coercion from a type $A$ into a type $B$ but also one from $B$ into $A$. So there are coercions from univariate polynomials in sparse representation over some ring to ones in dense representation and vice versa. Or we have

$$\mathtt{FF}(t_{\mathsf{integral\_domain}}) \trianglelefteq \mathtt{FF}(\mathtt{FF}(t_{\mathsf{integral\_domain}}))$$

and

$$\mathtt{FF}(\mathtt{FF}(t_{\mathsf{integral\_domain}})) \trianglelefteq \mathtt{FF}(t_{\mathsf{integral\_domain}}).$$

Other examples can be found in Sec. 8.2.5. If $A \trianglelefteq B$ and $B \trianglelefteq A$ then we will write $A \trianglerighteq\!\trianglelefteq B$.

If we require that for coercions

$$\phi : A \longrightarrow B,$$
$$\psi : B \longrightarrow A$$

the compositions $\phi \circ \psi$ and $\psi \circ \phi$ are the identities on $A$ resp. $B$, then the coherence theorem 6 can be extended to the case of type isomorphisms.[42]

So type isomorphisms can be seen as equivalence classes in the preorder on types induced by the coercions. However, there are several reasons to treat type isomorphisms by a new typing construct independent from the concept of coercions. As we have shown in Sec. 8.2.5 there is usually the second-order type of a category present in Axiom for a class of equivalent types. On the one hand if coercions are present in the system the equivalence classes in the coercion preorder can be deduced by a system so that it is not necessary to define them by the programmer.[43] On the other hand — at least for the purpose of a user interface — it seems to be useful to have a class of isomorphic types present as a first-order type. Since all equivalence classes in the coercion preorder are finite — only finitely many (possibly polymorphic) functions can be defined to be coercions — the type of finite disjoint unions — variant record types — can serve as a well known first-order type for that purpose (cf. [That91, p. 46]).

Moreover, it is reasonable to assume that type isomorphisms have the following properties which cannot be deduced from the properties of general coercion functions.

1. Isomorphic types belong to the same type class, i. e. if $t_1 : \sigma$ and $t_1 \underline{\bowtie} t_2$ then $t_2 : \sigma$.
2. If $f : (\sigma_1 \cdots \sigma_n)\sigma$ is an $n$-ary type constructor, $t_1 : \sigma_1, \ldots, t_n : \sigma_n$, $t'_1 : \sigma_1, \ldots, t'_n : \sigma_n$, such that
$$t_i \underline{\bowtie} t'_i \quad \forall i$$
then
$$f(t_1, \ldots, t_n) \underline{\bowtie} f(t'_1, \ldots, t'_n).$$

The second condition is only implied by the rules for structural coercions if $f$ would be monotonic or antimonotonic in all arguments. Because of the second condition a *congruence relation* is defined by $\underline{\bowtie}$ on the term-algebra of types.[44] Thus we can built the factor algebra modulo this congruence relation. This factor algebra is isomorphic to the factor algebra modulo some equational theory, the equational theory which is obtained if we interpret $\underline{\bowtie}$ as equality. We will call this equational theory *the equational theory corresponding to the type isomorphism.*

For simplicity we will often neglect the sort constraints but will only write the unsorted part. Since for many examples in consideration the sort is always the same, these slightly sloppy view can be justified even formally.

While it is useful to know that certain *different types* are isomorphic — such as the sparse and dense representations of polynomials — there are other cases where it seems to be more appropriate to have a semantics of the type system implying that certain types are actually *equal*.

So the type system is not coherent if we define all naturally occurring embedding functions to be coercions and if we regard two types

$$\texttt{direct\_sum}(t_1, t_2) \text{ and } \texttt{direct\_sum}(t_2, t_1)$$

as being different. This example would not violate the coherence of the type system if we had not only two possible coercion functions implying that these types are isomorphic but

[42] Obviously, the conditions that $\phi$ and $\psi$ are true inverses of each other is also a necessary condition for coherence.
[43] In Axiom the isomorphic types are treated independently of the coercions.
[44] It follows from the properties of $\trianglelefteq$ alone that $\underline{\bowtie}$ defines an equivalence relation.

if these types are actually *equal* in the system. Notice that an implementation of this type constructor having these properties is possible. One just has to use the same techniques as are used for the representation of general associative and commutative operators in certain term-rewriting systems (see e. g. [Bund93, Sec. 10], [Bund93a]), i. e. a certain ordering on terms has to be given and the terms have to be represented in a *flattened form*.

In Sec. 8.3.4 we will give a family of type isomorphisms whose corresponding equational theory is not finitely axiomatizable. Thus all of these isomorphisms cannot be modeled by declaring finitely many functions to be coercions between types (even if we allow "polymorphic" coercion functions between polymorphic types). So these type isomorphisms could be only modeled in the system by a direct mechanism implying that certain types are equal.

### Independence of the Coercion Preorder from the Hierarchy of Type Classes

If two types are isomorphic, then they belong to the same type class.

Such a conclusion is not justified if there is only a coercion form $A$ into $B$. Consider for instance a field $K$. Its elements can be coerced to the constant polynomials in $K[x]$. Of course, the ring of polynomials over some field is no longer a field.

However, it cannot be concluded in general that $A \trianglelefteq B$ and $A : \sigma$ implies $B : \tau$ for some $\sigma \leq \tau$. Just the opposite holds for many important examples!

Consider e. g. the coercion from an integral domain into its field of fractions which is not only an integral domain but even a field. Similarly, any field can be embedded in its algebraic closure, i. e. in a structure which has additional "nice" properties, namely that it is an algebraically closed field. The constructions of the real numbers $\mathbb{R}$ or of $p$-adic completions of $\mathbb{Q}$ can be seen similarly. The field of rational numbers $\mathbb{Q}$ can be embedded in these structures — and is usually identified with its image under this embedding — which are complete metric spaces, a property that the original structure did not have.

The construction of structures which have additional "nice" properties and in which the original structure can be embedded is an important tool for mathematical reasoning.[45] Usually, the original structures and their images under this embedding are not distinguished notationally.

So the possibility to have coercions which induce a preorder on types that is quite independent on the preorder on types induced by the inheritance hierarchy on type classes seems to be important. Notice that these preorders would still differ even if we had allowed more sophisticated inheritance possibilities on type classes than the ones given in Axiom or Haskell. There have to be (at least) two hierarchies. The one corresponding to some form of "inheritance": more special structures (such as a "rings") inherit all properties of more general ones (such as "groups"), and another one reflecting possible embeddings of a structure into another that might have stronger properties.

**Remark** Of course, it is desirable to have some form of control over the possibilities how coercions behave with respect to the hierarchy on type classes. This seems to be possible.

All of the examples given above can be described by an unary type constructor $F$ such that

[45] The author could easily list several examples of such constructions from the area of mathematics he has worked on. Since this area is non-constructive we will omit them. However, it seems to be possible to find some examples in almost *any* area of mathematics.

for any types $A$ and $B$ of an appropriate sort the following holds:

$$\begin{array}{l}\text{If } A \trianglelefteq B, \text{ then } F(A) \trianglelefteq F(B),\\ A \trianglelefteq F(A),\\ F(F(A)) \bowtie F(A).\end{array}$$

Thus — if we interpret $\trianglelefteq$ as $\subseteq$ and $\bowtie$ as equality — the type constructor $F$ has the properties of a *closure operator* (see e. g. [Dave90],[Laue82]).

So the requirement that a type unary constructor which has a direct embedding and whose constructed type belongs to a type class with stronger properties then the type parameter has to be a closure operator in the sense of above would be fulfilled by many important examples. On the other hand such a restriction might allow much more efficient type inference algorithms so that it might be a reasonable requirement for a system.

### Some Problematic Examples of Type Isomorphisms

In this section we will collect some natural examples of type isomorphisms which arise in the context of computer algebra. We will show that their corresponding equational theories are not unitary or even not finitary unifying or that the unification problem is even undecidable.

In Sec. 8.3.7 we will show why these properties of the corresponding equational theory are problematic in the context of type inference.

We have already shown that a family of type isomorphisms whose corresponding equational theory is not finitely axiomatizable cannot be modeled by means of finitely many coercion functions and thus requires another concept. The presentation of a family of type isomorphisms having this property will be given in the next section because the proof of this property will need a little technical machinery.

**Example 1.** As was mentioned above for the type constructor `direct_sum` on Abelian groups the type isomorphisms

$$\texttt{direct\_sum}(t_1, t_2) \bowtie \texttt{direct\_sum}(t_2, t_1),$$

and

$$\texttt{direct\_sum}(t_1, \texttt{direct\_sum}(t_2, t_3)) \bowtie \texttt{direct\_sum}(\texttt{direct\_sum}(t_1, t_2), t_3)$$

hold.

Thus `direct_sum` would give rise to an equational theory modulo an associate and commutative operator. The unification problem for such an equational theory is decidable, but not unitary unifying. However, it is finitary unifying (cf. [Siek89], [Joua90]).

**Example 2.** For the binary type constructor `pair` which builds the type of ordered pairs of elements of arbitrary types the following type isomorphisms hold:

$$\texttt{pair}(\texttt{pair}(A, B), C) \bowtie \texttt{pair}(A, \texttt{pair}(B, C)),$$

i. e. it corresponds to an associative equational theory. Unification for such theories is decidable but not finitary unifying [Siek89].

**Example 3.** Let $A, B, C$ be vector spaces over some fixed field $K$ and let $\oplus$ denote the direct sum of vector spaces and $\otimes$ denote the tensor product of two vector spaces. Then we have

$$(A \oplus B) \otimes C \cong (A \otimes C) \oplus (B \otimes C)$$

| Type isomorphisms whose corresponding equational theory | Example given on page |
|---|---|
| is not unitary unifying | 188 |
| is not finitary unifying | 188 |
| has an undecidable unification problem | 188 |
| is not finitely axiomatizable | 188–191 |

Figure 8.9: Some problematic examples of type isomorphisms

(see e. g. [Kowa63, p. 293].) Thus if we had two binary type constructors over vector spaces building direct sums and tensor products respectively, then the "distributivity law" gives rise to type isomorphisms. Since associativity and commutativity also hold for the type constructor building direct sums of vector spaces alone — any vector space is an Abelian group — we have the case of an equational theory having two operators obeying associativity, commutativity, and distributivity but no other equations.

Unfortunately, unification for such theories is undecidable [Siek89], [Szab82].

### 8.3.4 A Type Coercion Problem

In this section we want to present an example of a family of types which allow type-isomorphisms which correspond to an equational theory that is not finitely axiomatizable. In order to set up the example we first need a technical result.

#### A Technical Result

**Definition 23.** *Let $f : \{P, F\}^* \longrightarrow \{P, F\}^*$ be the function, which is defined by the following algorithm:*

> *If no $F$ is occurring in the input string, then return the input string as output string.*
>
> *Otherwise, remove any $F$ except the leftmost occurrence from the input string and return the result as output string.*

*Let $\equiv$ be the binary relation on $\{P, F\}^*$ which is defined by*

$$\forall v, w \in \{P, F\}^* : \; v \equiv w \iff f(v) = f(w).$$

Obviously, the function $f$ can be computed in linear time and the relation $\equiv$ is an equivalence relation on $\{P, F\}^*$.

Let $\Sigma$ be the first-order signature consisting of the two unary function Symbols $F$ and $P$. We will now lift the equivalence relation $\equiv$ to a set of equations over $\Sigma$.

**Definition 24.** *Let $\mathcal{E}$ be the following set of equations:*

$$\mathcal{E} = \{ \quad S_1(S_2(\cdots S_k(x)\cdots) = S_{k+1}(S_{k+2}(\cdots S_r(x)\cdots)) \mid \\ S_i \in \{F, P\}\ (1 \le i \le r) \text{ and } S_1 S_2 \cdots S_k \equiv S_{k+1} S_{k+2} \cdots S_r \quad \}$$

**Theorem 7.** $\mathcal{E}$ is not finitely based, i. e. there is no finite set of axioms for $\mathcal{E}$.

**Proof** Assume towards a contradiction that there is such a finite set $\mathcal{E}_0$. Let $\mathcal{M}$ be the free model of $\aleph_0$ generators over $\mathcal{E}$ and let $\mathcal{M}_0$ be the free model of one generator over $\mathcal{E}_0$.

Except for a possible renaming of the variable symbol $x$, $\mathcal{E}_0$ has to be a subset of $\mathcal{E}$. Otherwise, $\mathcal{E}_0$ would contain an equation of the form

$$S_1(S_2(\cdots S_k(x)\cdots) = S_{k+1}(S_{k+2}(\cdots S_r(y)\cdots)),$$

or of the form

$$S_1(S_2(\cdots S_k(x)\cdots) = S_{k+1}(S_{k+2}(\cdots S_r(x)\cdots)),\ \ S_1S_2\cdots S_k \not\equiv S_{k+1}S_{k+2}\cdots S_r.$$

However, none of these equations holds in $\mathcal{M}$.

Now let $n \in \mathbb{N}$ be the maximal size of a term in $\mathcal{E}_0$. Then the equation

$$F(\underbrace{P(P(\cdots(P}_{n}(x))\cdots))) = F(P(F(\underbrace{P(P(\cdots(P}_{n-1}(x))\cdots)))))$$

holds in $\mathcal{M}$, but it does not hold in $\mathcal{M}_0$. □

**The Problem**

If $R$ is an integral domain, we can form the field of fractions $\mathtt{FF}(R)$. We can also built the ring of univariate polynomials in the indeterminate $x$ which we will denote by $\mathtt{UP}(R, x)$ — the ring of polynomials $R[x]$ in the standard mathematical notation — which is again an integral domain by a Lemma of Gauß. Thus we can also built the field of fractions of $\mathtt{UP}(R, x)$, $\mathtt{FF}(\mathtt{UP}(R, x))$ — the field of rational functions $R(x)$.

Starting from an integral domain $R$ we will always get an integral domain and can repeatedly built the field of fractions and the ring of polynomials in a "new" indeterminate.

Thus if a computer algebra system has a fixed integral domain $\mathtt{R}$ and names for symbols $\mathtt{x}_0, \mathtt{x}_1, \mathtt{x}_2 \ldots$, it should also provide types of the form

1. $\mathtt{R}$,
2. $\mathtt{FF}(\mathtt{R})$,
3. $\mathtt{UP}(\mathtt{R}, \mathtt{x}_0)$,
4. $\mathtt{UP}(\mathtt{FF}(\mathtt{R}), \mathtt{x}_0)$,
5. $\mathtt{FF}(\mathtt{UP}(\mathtt{R}, \mathtt{x}_0))$,
6. $\mathtt{UP}(\mathtt{UP}(\mathtt{R}, \mathtt{x}_0), \mathtt{x}_1)$,
7. $\mathtt{UP}(\mathtt{FF}(\mathtt{UP}(\mathtt{R}, \mathtt{x}_0)), \mathtt{x}_1)$,
8. $\mathtt{FF}(\mathtt{UP}(\mathtt{UP}(\mathtt{R}, \mathtt{x}_0), \mathtt{x}_1))$,
9. $\mathtt{FF}(\mathtt{UP}(\mathtt{FF}(\mathtt{UP}(\mathtt{R}, \mathtt{x}_0)), \mathtt{x}_1)$,
10. $\mathtt{UP}(\mathtt{UP}(\mathtt{UP}(\mathtt{R}, \mathtt{x}_0), \mathtt{x}_1), \mathtt{x}_2)$,

    ⋮

It is convenient to use the same symbols for a mathematical object and the symbolic expression which denotes the object. In order to clarify things we will sometimes use additional $\langle\langle\cdot\rangle\rangle$ for the mathematical objects.

There are canonical embeddings from an integral domain into its field of fractions and into the ring of polynomials in one indeterminate (an element is mapped to the corresponding constant polynomial).

It is common mathematical practice to identify the integral domain with its image under these embeddings. Thus the type system should also provide a coercion between these types, i. e. if $t$ is a type variable of sort integral_domains and $x$ is of sort symbol, then

$$t \trianglelefteq \mathtt{FF}(t)$$

and

$$t \trianglelefteq \mathtt{UP}(t, x).$$

However, not all of the types built by the type constructors FF and UP should be regarded to be different. If the integral domain $R$ happens to be a field, then $R$ will be isomorphic to its field of fractions. Especially, for any integral domain $R$, $\langle\!\langle \mathtt{FF}(R) \rangle\!\rangle$ and $\langle\!\langle \mathtt{FF}(\mathtt{FF}(R)) \rangle\!\rangle$ are isomorphic.

The fact that also $\langle\!\langle \mathtt{FF}(\mathtt{FF}(R)) \rangle\!\rangle$ can be embedded in $\langle\!\langle \mathtt{FF}(R) \rangle\!\rangle$ can be expressed by

$$\mathtt{FF}(\mathtt{FF}(t)) \trianglelefteq FF(t),$$

which is one of the examples given in [Como91, p. 354].

But there are more isomorphisms which govern the relations of this family of types.

If we assume that an application of the type constructor UP always uses a "new" indeterminate as its second argument, any application of the type constructor FF except the outermost one application is redundant.

This observation will be captured by the following formal treatment. In order to avoid the technical difficulty of introducing "new" indeterminates, we will use an unary type constructor up instead the binary UP. The intended meaning of $\mathtt{up}(t)$ is $\mathtt{UP}(t, \mathtt{x}_n)$, where $\mathtt{x}_n$ is a new symbol, i. e. not occurring in $t$.

**Definition 25.** *Define a function* trans *from* $\{F, P\}^*$ *into the set of types recursively by the following equations. For* $w \in \{F, P\}^*$,

- $\mathsf{trans}(\varepsilon) = \mathtt{R}$,
- $\mathsf{trans}(Fw) = \mathtt{FF}(\mathsf{trans}(w))$,
- $\mathsf{trans}(Pw) = \mathtt{up}(\mathsf{trans}(w))$.

If we take $\langle\!\langle \mathtt{R} \rangle\!\rangle$ to be the ring of integers, the following lemma will be an exercise in elementary calculus.[46]

**Lemma 7.** *Let* $\langle\!\langle \mathtt{R} \rangle\!\rangle$ *be the ring of integers. For any* $v, w \in \{F, P\}^*$, *the integral domains* $\langle\!\langle \mathsf{trans}(v) \rangle\!\rangle$ *and* $\langle\!\langle \mathsf{trans}(w) \rangle\!\rangle$ *are isomorphic iff* $v \equiv w$.

*Moreover,* $\langle\!\langle \mathsf{trans}(v) \rangle\!\rangle$ *can be embedded in* $\langle\!\langle \mathsf{trans}(w) \rangle\!\rangle$ *and* $\langle\!\langle \mathsf{trans}(w) \rangle\!\rangle$ *can be embedded in* $\langle\!\langle \mathsf{trans}(v) \rangle\!\rangle$ *iff* $\langle\!\langle \mathsf{trans}(v) \rangle\!\rangle$ *and* $\langle\!\langle \mathsf{trans}(w) \rangle\!\rangle$ *are isomorphic.*

**Theorem 8.** *Let* $\Sigma$ *be the signature consisting of the unary function symbols* FF *and* up *and the constant* R. *Let* $\langle\!\langle \mathtt{R} \rangle\!\rangle$ *be the ring of integers.*

*Then there is no finite set of Equations* $\mathcal{E}'$ *over* $\Sigma$, *such that for ground terms* $t_1$ *and* $t_2$ *the following holds.*

$$\mathcal{E}' \models \{t_1 = t_2\} \iff \langle\!\langle t_1 \rangle\!\rangle \text{ and } \langle\!\langle t_2 \rangle\!\rangle \text{ are isomorphic.}$$

[46] If we started with the ring of polynomials in infinitely many indeterminates over some domain, then there would be additional isomorphisms.

**Proof** If $t_1$ and $t_2$ are ground terms, then there are $v, w \in \{F, P\}^*$ such that $t_1 = \mathsf{trans}(v)$ and $t_2 = \mathsf{trans}(w)$. Now we are done by Lemma 7 and Theorem 7. □

The problem is that the equational theory which describes the coercion relations in the example we gave is not finitely based. Since this property of an equational theory is *equivalence-invariant* in the sense of [Grae79, p. 382], the use of another signature for describing the types does not help.

### 8.3.5 Properties of the Coercion Preorder

If the type system is coherent, then the category of ground types as objects and the coercions as arrows is a preorder. Even if the type system is not coherent, a reflexive and transitive relation on the ground types (and even on the polymorphic types) is defined by "$\trianglelefteq$", i. e. a preorder.[47]

Factoring out the equivalence classes of this reflexive and transitive relation we will obtain a partial order on the types.

In general this order on the types will not be a lattice if we consider some typical examples occurring in a computer algebra system. Take e. g. the types `integer` and `boolean`. There is no type which can be coerced to both of these types (unless an additional "empty type" is present in the system).

For many purposes, especially type inference (see Sec. 8.3.7), it would be convenient if this partial ordering on the types were a quasi-lattice. In the following we will show that in general this will not be the case.

**Example 4.** Let `I` be the ring of integers and let $\oplus$ denote the direct sum of two Abelian groups and let the direct embeddings into the first argument and into the second argument of this type constructor be present, i. e. $\mathcal{D}_\oplus = \{1, 2\}$. Then we have

$$\begin{aligned}
&\mathtt{UP}(\mathtt{I}, \mathtt{x}) \trianglelefteq \mathtt{UP}(\mathtt{FF}(\mathtt{I}), \mathtt{x}),\\
&\mathtt{UP}(\mathtt{I}, \mathtt{x}) \trianglelefteq \mathtt{UP}(\mathtt{I}, \mathtt{x}) \oplus \mathtt{FF}(\mathtt{I}),\\
&\mathtt{FF}(\mathtt{I}) \trianglelefteq \mathtt{UP}(\mathtt{FF}(\mathtt{I}), \mathtt{x}),\\
&\mathtt{FF}(\mathtt{I}) \trianglelefteq \mathtt{UP}(\mathtt{I}, \mathtt{x}) \oplus \mathtt{FF}(\mathtt{I}),
\end{aligned}$$

and no other coercions can be defined between these types. There is also no type $R$ with $R \neq \mathtt{UP}(\mathtt{I}, \mathtt{x})$ and $R \neq \mathtt{FF}(\mathtt{I})$ such that

$$\begin{aligned}
&R \trianglelefteq \mathtt{UP}(\mathtt{FF}(\mathtt{I}), \mathtt{x}),\\
&R \trianglelefteq \mathtt{UP}(\mathtt{I}, \mathtt{x}) \oplus \mathtt{FF}(\mathtt{I})
\end{aligned}$$

(cf. Fig. 8.10). Thus in this case the partial ordering given by $\trianglelefteq$ is not a quasi-lattice (see also Lemma 3).

Even if we require $|\mathcal{D}_f| \leq 1$ for all type constructors — recall that this requirement is also necessary in order to ensure a coherent type system — and we have only direct embeddings and structural coercions then it is still possible that the partial ordering on types induced by "$\trianglelefteq$" is not a quasi-lattice. Consider for instance two type constructors $f : (\sigma)\sigma$ and $g : (\sigma)\sigma$ which we assume to be unary for simplicity. If $\mathcal{D}_f \cap \mathcal{M}_f \neq \emptyset$ and $\mathcal{D}_g \cap \mathcal{M}_g \neq \emptyset$ and $t : \sigma$, then

$$g(t) \trianglelefteq f(g(t)) \qquad \text{and} \qquad g(t) \trianglelefteq g(f(t))$$

[47] Notice the difference between *a category which is a preorder* and *a relation which is a preorder*.

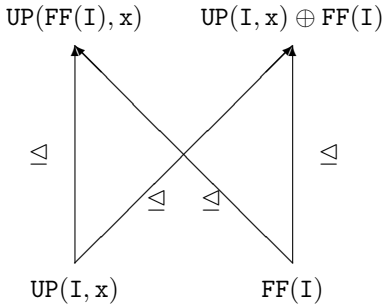



Figure 8.10: Ad Example 4

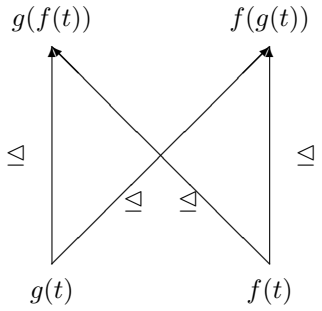



Figure 8.11: Another counter-example for the coercion order

and similarly

$$f(t) \trianglelefteq g(f(t)) \qquad \text{and} \qquad f(t) \trianglelefteq f(g(t))$$

(cf. Fig. 8.11). Having only direct embeddings and structural coercions the condition imposed in Lemma 3 with $a = g(t)$, $b = f(t)$, $c = f(g(t))$ and $d = g(f(t))$ are fulfilled.

The type constructors `FF` and `up` have such properties. However, we can define

$$\texttt{up}(\texttt{FF}(R)) \trianglelefteq \texttt{FF}(\texttt{up}(R))$$

for any integral domain $R$ using a coercion which is not a direct embedding nor a structural coercion.

So in this case some "ad hoc knowledge" can be used to avoid that the partial ordering induced by $\trianglelefteq$ is not a quasi-lattice.

In general, it does not seem to be justified to assume that the partial ordering induced by $\trianglelefteq$ is a quasi-lattice.

### 8.3.6 Combining Type Classes and Coercions

Let

$$\mathrm{op} : \overbrace{v_\sigma \times \cdots \times v_\sigma}^{n} \longrightarrow v_\sigma$$

be an $n$-ary operator defined on a type class $\sigma$ and let $A \trianglelefteq B$ be types belonging to $\sigma$ and let

$$\phi : A \longrightarrow B$$

be the coercion function. Moreover, let $\mathrm{op}_A$ and $\mathrm{op}_B$ be the instances of op in $A$ resp. $B$.

For $a_1, \ldots, a_n \in A$ the expression

$$\mathrm{op}(a_1, \ldots, a_n)$$

might denote different objects in $B$, namely

$$\mathrm{op}_B(\phi(a_1), \ldots, \phi(a_n))$$

or

$$\phi(\mathrm{op}_A(a_1, \ldots, a_n)).$$

The requirement of a unique meaning of

$$\mathrm{op}(a_1, \ldots, a_n)$$

just means that $\phi$ has to be a *homomorphism* for $\sigma$ with respect to op.

The typing of op in the example above is only one of several possibilities. In general if $\sigma$ is a type class having $p_{\tau_1}, \ldots, p_{\tau_k}$ as parameters — i. e. $p_{\tau_i}$ is a type variable of sort $\tau_i$ — then a $n$-ary first-order operation op defined in $\sigma$ can have the following types.[48]

$$\mathrm{op} : \xi_1 \times \cdots \times \xi_n \longrightarrow \xi_{n+1},$$

where $\xi_i$, $1 \leq i \leq n+1$, is either $v_\sigma$, or $p_{\tau_l}$, $l \leq k$, or a ground type $t_m$.

As on page 180 let $\mathcal{C}_\sigma$ be the category of ground types of sort $\sigma$ as objects and the coercions as arrows. For a ground type $t$ let $\mathcal{C}_t$ be the subcategory which has $t$ as single object and has thus the identity on $t$ as single arrow.[49] Now let

$$\mathcal{C}_i = \begin{cases} \mathcal{C}_\sigma, & \text{if } \xi_i = v_\sigma, \\ \mathcal{C}_{\tau_l}, & \text{if } \xi_i = p_{\tau_l}, \\ \mathcal{C}_{t_m}, & \text{if } \xi_i = t_m \text{ for a ground type } t_m. \end{cases}$$

Let $\Gamma_{\mathrm{op}}$ be a functor from $\mathcal{C}_1 \times \cdots \times \mathcal{C}_n$ into $\mathcal{C}_{n+1}$. If $(\zeta_1, \ldots, \zeta_n)$ is an object of $\mathcal{C}_1 \times \cdots \times \mathcal{C}_n$, i. e.

$$\zeta_i = \begin{cases} A_\sigma, & \text{if } \xi_i = v_\sigma \text{ and } A_\sigma \text{ is a ground type belonging to } \sigma, \\ A_{\tau_l}, & \text{if } \xi_i = p_{\tau_l} \text{ and } A_{\tau_l} \text{ is a ground type belonging to } \tau_l, \\ t_m, & \text{if } \xi_i = t_m, \end{cases}$$

then $\Gamma_{\mathrm{op}}(\zeta_1, \ldots, \zeta_n)$ is an object of $\mathcal{C}_{n+1}$, i. e. a ground type belonging to $\sigma$ resp. $\tau_{l'}$, or is a ground type $t_{m'}$ depending on the value of $\xi_{n+1}$.

[48] For simplicity, we will exclude in the following discussion arbitrary polymorphic types different from type variables. Especially, we will not regard higher-order functions, which do not play a central role in computer algebra although they are useful, cf. Sec. 8.2.2. For the other relevant cases of polymorphic types the following can be generalized easily.

[49] If the type system is not coherent this subcategory might have more than one arrow.

Informally $\Gamma_{\text{op}}$ can be used to specify the type of the range of an instantiation of op if instantiations of $\sigma$ and the parameters of $\sigma$ are given. We need a functor $\Gamma_{\text{op}}$ because of the following reason. Given two instantiations of the type class which can be described by $(\zeta_1, \ldots, \zeta_n)$ and $(\zeta_1', \ldots, \zeta_n')$ such that

$$\zeta_i \trianglelefteq \zeta_i' \quad \forall i \leq n$$

it is necessary that

$$\Gamma_{\text{op}}(\zeta_1, \ldots, \zeta_n) \trianglelefteq \Gamma_{\text{op}}(\zeta_1', \ldots, \zeta_n').$$

Otherwise, if $a_i$ is an object of type $\zeta_i$, $1 \leq i \leq n$, the expression

$$\text{op}(a_1, \ldots, a_n)$$

has the types $\Gamma_{\text{op}}(\zeta_1, \ldots, \zeta_n)$ and $\Gamma_{\text{op}}(\zeta_1', \ldots, \zeta_n')$ for which a coercion has to be defined in order to give the expression a unique meaning.

If $\sigma$ is a non-parameterized type class *any* mapping assigning an appropriate type to a tuple $(\zeta_1, \ldots, \zeta_n)$ can be extended to a functor. So the requirement that $\Gamma_{\text{op}}$ is a functor is only a restriction for parameterized type classes.

Since in a coherent type system there are unique coercions between types, we will omit the names of the coercions in the following and we will write

$$\Gamma_{\text{op}}(\zeta_1 \trianglelefteq \zeta_1', \ldots, \zeta_n \trianglelefteq \zeta_n')$$

for the image of the single arrow between the objects

$$(\zeta_1, \ldots, \zeta_n) \text{ and } (\zeta_1', \ldots, \zeta_n')$$

in the category

$$\mathcal{C}_1 \times \cdots \times \mathcal{C}_n$$

under the functor $\Gamma_{\text{op}}$. Thus $\Gamma_{\text{op}}(\zeta_1 \trianglelefteq \zeta_1', \ldots, \zeta_n \trianglelefteq \zeta_n')$ is an arrow in $\mathcal{C}_{n+1}$.

Let $\mathsf{SET}$ be the category of all set as objects and functions as arrows.[50]

By the assumption of set theoretic ground types and coercion functions we can assign to any object of $\mathcal{C}_\sigma$ an object of $\mathsf{SET}$ and to any arrow in $\mathcal{C}_\sigma$ an arrow of $\mathsf{SET}$ in a functorial way. We will write $\mathbf{T}_{\mathcal{C}_\sigma}$ for the functor defined by this mapping.

We will use the notation $\zeta_i \trianglelefteq \zeta_i'$ to denote the single arrow between $\zeta_i$ and $\zeta_i'$ in $\mathcal{C}_i$. Thus

$$\mathbf{T}_{\mathcal{C}_1} \times \cdots \times \mathbf{T}_{\mathcal{C}_n}(\zeta_1 \trianglelefteq \zeta_1', \ldots, \zeta_n \trianglelefteq \zeta_n')$$

is an arrow in the category

$$\underbrace{\mathsf{SET} \times \cdots \times \mathsf{SET}}_{n}.$$

Since $n$-tuples of sets are sets there is a functor from $\mathsf{SET}^n$ into $\mathsf{SET}$ which we will denote by $\mathbf{F}_n$.

If $(\zeta_1, \ldots, \zeta_n)$ is an object in $\mathcal{C}_1 \times \cdots \times \mathcal{C}_n$ we are now ready to formalize a requirement on the instantiation of op given by $(\zeta_1, \ldots, \zeta_n)$. We will not impose this condition directly on $\text{op}_{(\zeta_1, \ldots, \zeta_n)}$. It will be convenient to regard the set-theoretic interpretation

$$\mathbf{T}_{\mathcal{C}_1} \times \cdots \times \mathbf{T}_{\mathcal{C}_n}(\zeta_1, \ldots, \zeta_n)$$

[50] Notice that the category theoretic object $\mathsf{SET}$ is quite different from the Axiom category `SetCategory`.

of $(\zeta_1, \ldots, \zeta_n)$ instead this $n$-tuple of types itself. Then the set-theoretic interpretation of $\text{op}_{(\zeta_1,\ldots,\zeta_n)}$ induces a function between

$$\mathbf{F}_n(\mathbf{T}_{\mathcal{C}_1} \times \cdots \times \mathbf{T}_{\mathcal{C}_n}(\zeta_1, \ldots, \zeta_n))$$

and

$$\mathbf{T}_{\mathcal{C}_{n+1}}(\Gamma_{\text{op}}(\zeta_1, \ldots, \zeta_n)),$$

which we will denote by $\mathbf{O}_{\text{op}}(\zeta_1, \ldots, \zeta_n)$.

Given $(\zeta_1, \ldots, \zeta_n)$ and $(\zeta_1', \ldots, \zeta_n')$ such that

$$\zeta_i \trianglelefteq \zeta_i' \quad \forall i \leq n$$

we just need that the following diagram is commutative.

$$\begin{array}{ccc}
\mathbf{F}_n(\mathbf{T}_{\mathcal{C}_1} \times \cdots \times \mathbf{T}_{\mathcal{C}_n}(\zeta_1, \ldots, \zeta_n)) & \xrightarrow{\mathbf{O}_{\text{op}}(\zeta_1, \ldots, \zeta_n)} & \mathbf{T}_{\mathcal{C}_{n+1}}(\Gamma_{\text{op}}(\zeta_1, \ldots, \zeta_n)) \\
\Big\downarrow \text{L} & & \Big\downarrow \text{R} \\
\mathbf{F}_n(\mathbf{T}_{\mathcal{C}_1} \times \cdots \times \mathbf{T}_{\mathcal{C}_n}(\zeta_1', \ldots, \zeta_n')) & \xrightarrow[\mathbf{O}_{\text{op}}(\zeta_1', \ldots, \zeta_n')]{} & \mathbf{T}_{\mathcal{C}_{n+1}}(\Gamma_{\text{op}}(\zeta_1', \ldots, \zeta_n'))
\end{array}$$

In the diagram above we have set

$$\text{L} = \mathbf{F}_n(\mathbf{T}_{\mathcal{C}_1} \times \cdots \times \mathbf{T}_{\mathcal{C}_n}(\zeta_1 \trianglelefteq \zeta_1', \ldots, \zeta_n \trianglelefteq \zeta_n'))$$

and

$$\text{R} = \mathbf{T}_{\mathcal{C}_{n+1}}(\Gamma_{\text{op}}(\zeta_1 \trianglelefteq \zeta_1', \ldots, \zeta_n \trianglelefteq \zeta_n')).$$

This requirement on $\mathbf{O}_{\text{op}}$ can be read that $\mathbf{O}_{\text{op}}$ is a *natural transformation* between the functor

$$\mathbf{F}_n \circ (\mathbf{T}_{\mathcal{C}_1} \times \cdots \times \mathbf{T}_{\mathcal{C}_n})$$

and the functor

$$\mathbf{T}_{\mathcal{C}_{n+1}} \circ \Gamma_{\text{op}}.$$

Thus for a $n$-ary first-order operator op the requirements that

1. the assignments of a range type for an operation given instantiations of a type class and its parameters has to be "functorial" and
2. the instantiation of the operator has to correspond to a natural transformation between functors giving the set-theoretic interpretations of the ground types and the coercions between them

will guarantee that type classes and coercions interact nicely, i. e. give expressions involving op a unique meaning.

A brief inspection of the examples of parameterized type classes occurring in Axiom by the author has suggested that there is no example violating the first requirement which will always hold in non-parameterized type classes. Nevertheless, a formal requirement for a computer algebra language seems to be useful to ensure that no such violating will occur in future extensions.

The second requirement is formulated as one on the possible instantiations of operators. However, it can also be read that given the instantiations only certain coercions between base types are allowed, namely only coercions for which the interpretation is a natural transformation. We will show below that using this view we can conclude that only "injective" coercion functions are allowed between most types.[51]

**Remark** Our conditions imposed on the combination of type classes and coercions are an adaptation of the work of Reynolds [Reyn80] on *category-sorted algebras*. The difference is that Reynolds allows each operator to be generic, i. e. that it may be instantiated with any type in any position. We allow type-class polymorphism at some position and do not allow polymorphism at all in other positions which seems to be the natural way to describe many important examples.

### Injective Coercions

An important type class is the class of types on which a test for equality of objects can be performed in the system.[52] In this type class the operator symbol

$$= : t_{\mathsf{Eq}} \times t_{\mathsf{Eq}} \longrightarrow \mathtt{Boolean}$$

is used to denote the system test for equality. In order to distinguish between the "system equality" and "true equality" we will use

$$\mathtt{isequal} : t_{\mathsf{Eq}} \times t_{\mathsf{Eq}} \longrightarrow \mathtt{Boolean}$$

for the system equality in the following.

Then the boolean values of

$$\mathtt{isequal}(a_1, a_2)$$

and

$$\mathtt{isequal}(\phi(a_1), \phi(a_2))$$

have to be the same. Especially, if the latter evaluates to `true` then the former also has to evaluate to `true`. In analogy to the definition of injective this means that $\phi$ has to be an injective function "modulo system equality" (usually, the definition of injective involves true equality).

Thus coercions between types belonging to the "equality type class" have to be "injective."

The system equality for a type might very well differ from the equality defined on a certain data type representing it. So very often the rational numbers are just represented as pairs of integers. Then different pairs of integers can represent the same rational number, thus the system test for equality of rational numbers is different from the equality on pairs of integers.

Of course, a non-injective coercion function would not violate our requirements, if $A$ and $B$ do not use the same operator symbol as a test for equality. Thus defining two different type classes Eq1 and Eq2 with operators `isequal1` resp. `isequal2` as tests for equality and having $A$ of type class Eq1 and $B$ of type class Eq2 would allow to define a non-injective function to be a coercion between $A$ and $B$. Defining such different type classes is also a clear indication for the user that there are problems. Exposing a problem seems to be preferable than hiding

[51] In the following we will precisely state what we mean by "injective" and "most types."
[52] It is called Eq in Haskell and `SetCategory` in Axiom.

it and and hoping that it will not occur. Although usually for two elements $a_1$ and $a_2$ of type $A$ the test for equality in $A$ will be used and not the one in $B$ it might happen that one of the elements is coerced to $B$. Probably, this will not happen very frequently which makes the situation even more dangerous, since the system will wrongly say that two elements are equal only in situations which are rather complicated so that the behavior of the system might not be clear for the user.[53]

So the requirement of "injective" coercions seems to be absolutely necessary for a computer algebra system although it is not required by a system like Axiom![54]

### 8.3.7 Type Inference

In Sec. 8.2.2 we have seen that the type inference problem for a language having type classes is decidable even if we have a language with higher-order functions and one allowing parametric polymorphism. Moreover, there is a finite set of types for any object of the language such that any type of the object is a substitution instance of one of those types.

The type inference problem for a language with coercions is much more complicated. So there are objects which have infinitely many types which are not substitution instances of finitely many (polymorphic) types.[55] Consider a type `R` belonging to a type class commutative_ring and let $r$ be be an object of type `R`. Given coercions

$$v_{\mathsf{commutative\_ring}} \trianglelefteq \mathtt{up}(v_{\mathsf{commutative\_ring}})$$

then $r$ also has the types

$$\mathtt{up}(\mathtt{R}), \mathtt{up}(\mathtt{up}(\mathtt{R})), \ldots$$

In [Mitc91], [Fuhx89], [Fuhx90] type systems for functional languages allowing coercions between base types and structural coercions are given and type inference algorithms for them. These systems do not allow type class polymorphism nor parametric polymorphism. In [Brea89], [Brea91] a system having coercions and parametric polymorphism is given; however, no type inference for the system is provided.

In [That91] a type inference system for the case of type isomorphisms induced by coercions is given which allows parametric polymorphism. However, as is argued in [That91] if the equational theory corresponding to the type isomorphisms is not unitary unifying then the semantics of an expression involving `let` may be ambiguous. Moreover, the type inference problem is reduced to an unification problem over the equational theory corresponding to the type isomorphisms. So in the case of an undecidable equational unification problem (cf. Example 3) only a semi-decision method is available for type inference.

Type inference algorithms for a system allowing parametric polymorphism and records resp. variants are given in [Wand87], [Wand88], [Wand89], [Wand91], [Stan88], [Leis87], [Remy89].

[53] For instance, the situation described above arises when coercions between (arbitrary precision) integers and floating point numbers are defined and the same symbol is used as a test for equality. Then two integers $a$ and $b$ which are not equal might be equal if they are coerced to floating point numbers. Such a coercion is used in many system if an expression like "$a + 0.0$" occurs and can thus happen in situations which are quite surprising for the user.

[54] Since it is an undecidable problem to check whether a given recursive function is injective — which can be easily proved by applying Rice's Theorem — it is not possible to enforce by a compiler that coercions are injective if functions defined by arbitrary code can be declared to be coercions. Nevertheless, it seems to be useful to state this requirement as a guideline for a programmer.

[55] Using the results of Sec. 8.3.7 it will be possible to assign finitely many types to an object in the subsystem described in that section which have "minimal properties" among all types of the object.

Since variants can be used to model classes of isomorphic types some of these results can be applied if we model classes of isomorphic types as variants.

Kaes [Kaes92] gives a system allowing type-class polymorphism (also parametric type classes can be described) which can handle coercions between base types and structural coercions according to our definition.[56] However, direct embeddings are not allowed.

In [Como91] a type inference system and a semi-decision procedure for it are described. However, in that system some assumptions on the properties on coercions are imposed which are not justified for many examples occurring in computer algebra.[57] In [Como91] a proof is given that the type inference problem for the described system becomes undecidable if no restrictions on the coercions are imposed.

Since there are infinitely many ground types in a system usually infinitely many coercions will be necessary. However, with the exception of the example stated in Sec. 8.3.4 all examples of coercions we have given — such as the direct embeddings and the structural coercions — can be described by a finite set of Horn clauses which will usually have variables. The formalism of Horn clauses is strong enough to capture type classes and even parametric type classes and also polymorphic types can be described. Then the typability of an object can be stated as the question whether a certain clause is the logical consequence of the given set of Horn clauses. Thus using a complete Horn clause theorem prover[58] we have a semi-decision procedure for type inference. The size of the search space seems to be a problem for the practical use of this method, but not the fact that it is only a semi-decision procedure. If an expression cannot be typed using certain resources — i. e. a typing of the expression involves too many coercions if it is typeable at all — it does not seem to be a practical limitation if a system rejects the expression as possibly untypeable and asks the user to provide more typing information if the user thinks that the expression is typeable.

It is not clear which classes of coercions in connection with which other typing constructs are allowed such that the type inference problem is decidable. Coercions between polymorphic types are certainly a problem. In the following we will shortly discuss to what extent some restrictions are justified for a computer algebra system.

If type inference has to be performed for user defined functions, then polymorphic types arise naturally (cf. Sec. 8.2.2). Since the possibility to type user defined functions is useful for a computer algebra system but does not play the same central role as for a functional programming language it might be reasonable to exclude them from type inference if coercions are present in order to facilitate the problem.

But there are also other objects than functions that can be polymorphic. Especially there are naturally occurring examples of *polymorphic constants*.

In Haskell integer constants are polymorphic constants. If $n$ is a constant denoting an integer then it also denotes the corresponding objects of the types in the type class `Num`. Having a language allowing coercions the use of polymorphic constants can be avoided for the examples used in Haskell, because coercions can be defined between the types belonging to `Num` in Haskell.[59]

In a computer algebra system there are more types present which have objects usually

[56] In the systems in [Mitc91], [Fuhx90], [Fuhx89] all type constructors have to be monotonic or antimonotonic in all arguments.

[57] The problematic assumptions are that all type constructors have to be monotonic in all arguments and that any polymorphic type can be coerced to its substitution instances.

[58] Notice that PROLOG is not one because of the used depth-first search strategy.

[59] In Haskell only explicit conversions but no implicit coercions are allowed.

denoted by integer constants. A nice example showing the use of polymorphic constants in mathematical notation is given by Rector [Rect89, p. 304]:

> Consider
> $$\frac{(x+y)^{1+n}+1}{1+nx}$$
> where the user wants to work with rational functions over a finite field of $p$-elements. This formula presents the problem of polymorphic constants. To a mathematician, the types of each subexpression are immediately clear: $n$ is an integer variable which must be reduced modulo $p$ in the denominator of the expression, $x$ and $y$ are finite field variables, 1 appearing in the exponent is an integer and the other 1's are the multiplicative identity of the finite field."

Since there are no embeddings from $\mathbb{Z}$ into $\mathbb{Z}_m$ nor from $\mathbb{Z}_m$ into $\mathbb{Z}$ — for $n \neq m$ there is not even one from the ring $\mathbb{Z}_m$ into the ring $\mathbb{Z}_n$[60] the use of polymorphic constants cannot be avoided by introducing coercions.

### Algorithms for Type Inference

In the following section we will restrict the types to the ones which can be expressed as terms of a finite order-sorted signature. As we have seen in Sec. 8.2.1 we can also assume that the signature is regular.

Let op be a $n$-ary operation,

$$\mathrm{op} : \xi_1 \times \cdots \times \xi_n \longrightarrow \xi_{n+1},$$

where $\xi_i$, $1 \leq i \leq n+1$, is either a type variable $v_{\tau_l}$, $l \leq k$, or a ground type $\overline{t}_i$. Given objects $o_1, \ldots, o_n$ having types $t_1, \ldots, t_n$ respectively, the expression

$$\mathrm{op}(o_1, \ldots, o_n)$$

will be well typed having type $\xi_{n+1}$ iff the following conditions are satisfied.

1. If $\xi = \overline{t}_i$ for some ground type $\overline{t}_i$ then $t_i \trianglelefteq \overline{t}_i$.
2. If $\xi_i = \xi_j = v_{\tau_l}$ for some $i \neq j$ then there is a type $t : \tau_l$ such that $t_i \trianglelefteq t$ and $t_j \trianglelefteq t$.
3. If $\xi_i = v_{\tau_k}$ then there is a type $t : \tau_k$ such that $t_i \trianglelefteq t$.

Notice that if we require that all objects have ground types then algorithms solving the problems imposed by the above conditions can be used to solve the type inference problem using a bottom-up process.[61]

If we do not restrict the possible coercions then determining whether for given types $t_1$ and $t_2$ there is a type $t$ such that $t_1 \trianglelefteq t$ and $t_2 \trianglelefteq t$ might be an undecidable problem (cf. [Como91]).

In the following we will restrict the possible coercions to coercions between base types,[62] direct embeddings and structural coercions. In Sec. 8.3.2 we have defined the coercions only between ground types, because we have given semantic considerations on coercions and it

[60] If $n = km$ then there is an embedding of the *Abelian group* $\langle \mathbb{Z}_m, + \rangle$ into the Abelian group $\langle \mathbb{Z}_n, + \rangle$, namely the one given by the mapping $i \mapsto ki$. Notice that a declaration of this embedding to be a coercion between the corresponding types and to have the elements of $\mathbb{Z}$ as polymorphic constants (in their usual interpretation) in $\langle \mathbb{Z}_m, + \rangle$ and in $\langle \mathbb{Z}_n, + \rangle$ would contradict the requirements stated in Sec. 8.3.6.

[61] Similar ideas can be found in [Como91, Sec. 4] and in [Rect89].

[62] By the assumption of a finite signature there are only finitely many base types and we will assume that the finitely many coercions between base types are effectively given.

is not clear how to define a semantics for arbitrary polymorphic types. The algorithmic problems we are dealing with in this section can be seen as algorithmic problems on certain terms of an order-sorted signature where an additional relation "$\trianglelefteq$" is given. It will be convenient to define $\trianglelefteq$ also for polymorphic types, i. e. non-ground terms. It is clear how the definitions given in Sec. 8.3.2 for direct embeddings and structural coercions can be extended to polymorphic types.

We will assume that for any type constructor $f$ the set of direct embedding positions $\mathcal{D}_f$ and the sets $\mathcal{A}_f$ and $\mathcal{M}_f$ are well defined, i. e. independent of the arguments of $f$. Moreover, we will assume that for any types $t_1 \trianglelefteq t_2$ and any (sort-correct) substitution $\theta$ we also have $\theta(t_1) \trianglelefteq \theta(t_2)$. These assumptions are satisfied by all examples we gave and are natural for the formalism of describing types we use.

The advantage of extending the notions of direct embeddings and structural coercions to polymorphic types is that there are *finitely* many (polymorphic) types

$$t_1^1 \trianglelefteq t_1^2, \ldots, t_r^1 \trianglelefteq t_r^2$$

such that for any types $t_1 \trianglelefteq t_2$ there is a (sort-correct) substitution $\theta$ and an $1 \leq i \leq r$ such that

$$t_1 = \theta(t_i^1) \quad \text{and} \quad t_2 = \theta(t_i^2).$$

**Proposition 2.** *Assume that the types are terms of a finite, regular order-sorted signature and that there are only coercions between base types, direct embeddings and structural coercions. Then for any type $t$, the set*

$$\mathcal{S}_t = \{\sigma \mid \exists t' \,.\, t' : \sigma \text{ and } t \trianglelefteq t'\}$$

*is effectively computable.*

**Proof** We claim that the set $\mathcal{S}_t$ will be computed by $\mathsf{CSGT}(t)$ (see Fig. 8.12).

All computations which are used in CSGT and CSBT can be performed effectively. Since the signature is finite there are always only finitely many possibilities which have to be checked in the existential clauses of the algorithms and so algorithm CSBT will terminate and so will CSGT. Algorithm CSGT is correct (i. e. $\mathsf{CSGT}(t) \subseteq \mathcal{S}_t$), because only types and the sort of types $t$ can be coerced to are computed. Its completeness (i. e. $\mathsf{CSGT}(t) \supseteq \mathcal{S}_t$) follows from the fact that structural coercions cannot add new sorts to $\mathcal{S}_t$. □

In the following we will rule out antimonotonic structural coercions, i. e. we will require that $\mathcal{A}_f = \emptyset$ for all type constructors $f$.

Notice that the restriction $\mathcal{A}_f = \emptyset$ does not exclude type constructors like `FS` from the framework. Only the automatic insertion of a coercion giving rise to the antimonotony is excluded. For instance, instead of having `FS` as a type constructor which is antimonotonic in its first argument and monotonic in its second, it is one which is only monotonic in its second argument. Such a restriction does not seem to cause a loss of too much expressiveness. This is an important difference to the system in [Como91], in which all type constructors have to be monotonic in all arguments. Type constructors which are antimonotonic in some argument have to be excluded from that system in general, because it is not possible that a type constructor being antimonotonic in some argument can be made monotonic in that argument without changing the intended meaning of the type constructor. Thus our framework is more general in this respect than the one in [Como91]. However, direct embeddings are a special form of the "rewrite relations" for coercion considered in that

$\mathcal{S} \leftarrow \mathsf{CSGT}(t)$.

[Sorts of types a type $t$ is coercible to. $\mathcal{S}$ is the set of sorts of types in which $t$ can be coerced to. Assumes that the signature is finite, only direct embeddings and structural coercions are present.]

**(1)** [$t$ base type.] **if** $\mathrm{com}(t) = 0$ **then** $\{\, \mathcal{S} \leftarrow \mathsf{CSBT}(t)$; **return** $\}$.

**(2)** [Recurse.] Let $t = g(t_1, \ldots, t_m)$;
**for** $i = 1, \ldots, m$ **do** $\mathcal{S}_i \leftarrow \mathsf{CSGT}(t_i)$;
**for** $(\sigma_1, \ldots, \sigma_m) \in \mathcal{S}_1 \times \cdots \times \mathcal{S}_m$ **do** $\{$
**if** there is $g : (\sigma_1 \cdots \sigma_m)\overline{\sigma}$ such that $\overline{\sigma} \notin \mathcal{S}$ **then** $\{$
$\mathcal{T} \leftarrow \mathcal{T} \cup \{g(v_{\sigma_1}, \ldots, v_{\sigma_m})\}$; $\mathcal{S} \leftarrow \mathcal{S} \cup \{\overline{\sigma}\}$; $\mathcal{S}' \leftarrow \mathcal{S}$; $\mathcal{T}' \leftarrow \mathcal{T}$ $\}$ $\}$.

**(3)** [Compute Direct Embeddings.] **for** $\overline{t} \in \mathcal{T}$ **do** $\{$ **if** there are $\overline{\sigma}$, $f : (\sigma_1 \cdots \sigma_n)\sigma'$, $i \in \{1, \ldots, n\}$ such that $\overline{t} : \overline{\sigma}$ and $\sigma_i = \overline{\sigma}$ and $i \in \mathcal{D}_f$ and $\sigma' \notin \mathcal{S}$ **then** $\{$
$\mathcal{S}' \leftarrow \mathcal{S}' \cup \{\sigma'\}$; $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{f(v_{\sigma_1}, \ldots, v_{\sigma_n})\}$ $\}$ $\}$.

**(4)** [Iterate if something is added.] **if** $\mathcal{S}' \neq \mathcal{S}$ **then** $\{$
$\mathcal{S} \leftarrow \mathcal{S}'$; $\mathcal{T} \leftarrow \mathcal{T}'$; **goto (3)** $\}$.

where

$\mathcal{S} \leftarrow \mathsf{CSBT}(t)$.

[Sorts of types a base type $t$ is coercible to. $\mathcal{S}$ is the set of sorts of types in which $t$ can be coerced to. Assumes that the signature is finite, only direct embeddings and structural coercions are present.]

**(1)** [Initialize.] $\mathcal{T} \leftarrow \{t' \mid t \trianglelefteq t' \text{ and } t' \text{ is a base type}\}$;
$\mathcal{S} \leftarrow \{\sigma' \mid t : \sigma'\}$; $\mathcal{S}' \leftarrow \mathcal{S}$; $\mathcal{T}' \leftarrow \mathcal{T}$.

**(2)** [Compute Direct Embeddings.] **for** $\overline{t} \in \mathcal{T}$ **do** $\{$ **if** there are $\overline{\sigma}$, $f : (\sigma_1 \cdots \sigma_n)\sigma'$, $i \in \{1, \ldots, n\}$ such that $\overline{t} : \overline{\sigma}$ and $\sigma_i = \overline{\sigma}$ and $i \in \mathcal{D}_f$ and $\sigma' \notin \mathcal{S}$ **then** $\{$
$\mathcal{S}' \leftarrow \mathcal{S}' \cup \{\sigma'\}$; $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{f(v_{\sigma_1}, \ldots, v_{\sigma_n})\}$ $\}$ $\}$.

**(3)** [Iterate if something is added.] **if** $\mathcal{S}' \neq \mathcal{S}$ **then** $\{$
$\mathcal{S} \leftarrow \mathcal{S}'$; $\mathcal{T} \leftarrow \mathcal{T}'$; **goto (2)** $\}$.

Figure 8.12: Algorithms computing sorts of types a given type can be coerced to

paper. So the following can be seen as a solution for one of the open problems stated in [Como91], namely finding restrictions on the system of coercions which will yield a decidable type inference problem.

**Definition 26.** *If for two types $t_1$ and $t_2$ there is a type $t$ such that $t_1 \trianglelefteq t$ and $t_2 \trianglelefteq t$ then $t$ is called a* common upper bound *of $t_1$ and $t_2$.*

*A* minimal upper bound $\mathrm{mub}(t_1, t_2)$ *of two types $t_1$ and $t_2$ is a type $t$ satisfying the following conditions.*

1. *The type $t$ is a common upper bound of $t_1$ and $t_2$.*
2. *If $t'$ is a type which is a common upper bound of $t_1$ and $t_2$ such that $t' \trianglelefteq t$, then $t \trianglelefteq t'$.*

*A* complete set of minimal upper bounds *for two types $t_1$ and $t_2$ is a set* $\mathrm{CSMUB}(t_1, t_2)$ *such that*

1. *all* $t \in \mathrm{CSMUB}(t_1, t_2)$ *are a minimal common upper bound of* $t_1$ *and* $t_2$*, and*
2. *for every type* $t'$ *which is a common upper bound of* $t_1$ *and* $t_2$ *there is a* $t \in \mathrm{CSMUB}(t_1, t_2)$ *such that* $t \trianglelefteq t'$*.*

If two types $t_1$ and $t_2$ have no minimal upper bound then the complete sets of minimal upper bounds are all empty. In this case we will write $\mathrm{CSMUB}(t_1, t_2) = \emptyset$. We will write $|\,\mathrm{CSMUB}(t_1, t_2)|$ to denote the smallest cardinality of a complete set of minimal upper bounds of $t_1$ and $t_2$.

If the partial order induced by $\trianglelefteq$ is a quasi-lattice then $|\,\mathrm{CSMUB}(t_1, t_2)| \leq 1$ for all types $t_1$ and $t_2$. However, as we have seen in Sec. 8.3.5 this partial order will not be a quasi-lattice in general.

In the following we will assume that for any two *base types* $t_1^{\mathrm{b}}$ and $t_2^{\mathrm{b}}$ a *finite* complete set of minimal upper bounds can be computed effectively, say by $\mathsf{CSMUBBT}(t_1^{\mathrm{b}}, t_2^{\mathrm{b}})$. We will give an algorithm computing for any two types $t_1$ and $t_2$ a complete set of minimal upper bounds and will show that this set is finite.

**Theorem 9.** *Assume that all coercions are coercions between base types, direct embeddings and structural coercions. Moreover, assume that for all type constructors* $f$ *there is at most one direct embedding position, i. e.* $|\mathcal{D}_f| \leq 1$*, and no antimonotonic coercions are present, i. e.* $\mathcal{A}_f = \emptyset$*, and for any base types* $t_1^{\mathrm{b}}$ *and* $t_2^{\mathrm{b}}$ *there is a finite complete set of minimal upper bounds with respect to the set of base types which can be effectively computed by a function* $\mathsf{CSMUBBT}(t_1^{\mathrm{b}}, t_2^{\mathrm{b}})$*.*

*Then for any two types* $t_1$ *and* $t_2$ *there is a finite complete set of minimal upper bounds which can be effectively computed.*

**Proof** We claim that algorithm CSMUBGT (see Fig. 8.13) terminates for any input parameters $t_1$ and $t_2$ and computes a complete set of minimal upper bounds which is finite.

We will prove this claim by induction on the complexity of $t_1$ and $t_2$ along the steps of the algorithm.

If $t_1$ and $t_2$ are base types, then $\mathsf{CSMUBBT}(t_1, t_2)$ is also a complete set of minimal upper bounds of $t_1$ and $t_2$ with respect to all types. This subclaim can be proved by induction on the complexity of possible common upper bounds of $t_1$ and $t_2$ using the assumption that for any type constructor $f$ we have $|\mathcal{D}_f| \leq 1$.[63]

So the algorithm terminates for the case of base types and returns a finite set which is a complete set of minimal upper bounds for $t_1$ and $t_2$.

The algorithm will terminate for all other $t_1$ and $t_2$, too. Recursive calls of the algorithm are done on arguments of which at least one has a strictly smaller complexity. Since any of the recursive calls returns a finite set, only finitely many iterations have to be performed by the algorithm and the returned set is finite.

Since only direct embeddings and monotonic structural coercions are present, any element of $\mathcal{U}$ is a minimal upper bound of $t_1$ and $t_2$. The set $\mathcal{U}$ will be a complete set of minimal upper bounds, because $|\mathcal{D}_f| \leq 1$ for any type constructor and all other possibilities of minimal upper bounds for $t_1$ and $t_2$ are covered by the algorithm.

Since CSMUBGT returns a finite set of types, the existence of a finite set of minimal upper bounds follows from the correctness of the algorithm. □

**Remark** Since algorithm CSMUBGT uses the type constructors given by its arguments

[63] Without this assumption the subclaim is false in general.

$\mathcal{U} \leftarrow \mathsf{CSMUBGT}(t_1, t_2)$

[$\mathcal{U}$ is a complete set of minimal upper bounds of two types $t_1$ and $t_2$. Requires that only direct embeddings and structural coercions are used, $|\mathcal{D}_f| \leq 1$ and $\mathcal{A}_f = \emptyset$ for any type constructor $f$. Assumes that algorithm CSMUBBT returns a finite set.]

**(1)** [$t_1$ and $t_2$ base types.] **if** $\mathrm{com}(t_1) = 1$ and $\mathrm{com}(t_2) = 1$ **then** { $\mathcal{U} \leftarrow \mathsf{CSMUBBT}(t_1, t_2)$; **return** }.

**(2)** [Ensure that $\mathrm{com}(t_1) \leq \mathrm{com}(t_2)$.] **if** $\mathrm{com}(t_1) > \mathrm{com}(t_2)$ **then** { $h \leftarrow t_1$; $t_1 \leftarrow t_2$; $t_2 \leftarrow h$ }.

**(3)** [$t_1$ a base type.] **if** $\mathrm{com}(t_1) = 1$ **then** {
let $t_2 = f(t_2^1, \ldots, t_2^n)$;
**if** $|\mathcal{D}_f| = 0$ **then** { $\mathcal{U} \leftarrow \emptyset$; **return**};
let $\mathcal{D}_f = \{i\}$;
$\mathcal{U}' \leftarrow \mathsf{CSMUBGT}(t_1, t_2^i)$;

**(3.1)** **if** $\mathcal{U}' = \emptyset$ **then** { $\mathcal{U} \leftarrow \emptyset$; **return**};

**(3.2)** **if** $\mathcal{U}' \neq \emptyset$ **then** { **if** $i \in \mathcal{M}_f$ **then** { $\mathcal{U} \leftarrow \emptyset$;
**for** $t' \in \mathcal{U}'$ **do**
$\mathcal{U} \leftarrow \mathcal{U} \cup \{f(t_2^1, \ldots, t_2^{i-1}, t', t_2^{i+1}, \ldots, t_2^n)\}$ };
**if** $i \notin \mathcal{M}_f$ **then** { **if** $t_2^i \in \mathcal{U}'$ **then** $\mathcal{U} \leftarrow \{t_2\}$
**else** $\mathcal{U} \leftarrow \emptyset$} **return** } }.

**(4)** [General case.] let $t_1 = g(t_1^1, \ldots, t_1^m)$; let $t_2 = f(t_2^1, \ldots, t_2^n)$; $\mathcal{U} \leftarrow \emptyset$.

**(5)** [Structural coercions.] **if** $f = g$ **then** {
**for** $i \in \mathcal{M}_f$ **do** $\mathcal{U}_i \leftarrow \mathsf{CSMUBGT}(t_1^i, t_2^i)$;
let $\mathcal{M}_f = \{j_1, \ldots, j_l\}$;
**if** $t_1^k = t_2^k$ for all $k \in \{1, \ldots, n\} - \mathcal{M}_f$ **then** {
**for** $(t'_{j_1}, \ldots, t'_{j_l}) \in \mathcal{U}_{j_1} \times \cdots \times \mathcal{U}_{j_l}$ **do** {
**for** $k \in \{1, \ldots, n\} - \mathcal{M}_f$ **do** $t'_k \leftarrow t_1^k$;
$\mathcal{U} \leftarrow \mathcal{U} \cup \{f(t'_1, \ldots, t'_n)\}$ } } }.

**(6)** [Direct embeddings in $g$.] **if** $|\mathcal{D}_g| = 1$ **then** {
let $\mathcal{D}_g = \{i\}$; $\mathcal{U}' \leftarrow \mathsf{CSMUBGT}(t_1^i, t_2)$;
**if** $\mathcal{U}' \neq \emptyset$ **then** { **if** $i \in \mathcal{M}_g$ **then** { **for** $t' \in \mathcal{U}'$ **do** $\mathcal{U} \leftarrow \mathcal{U} \cup \{g(t_2^1, \ldots, t_2^{i-1}, t', t_2^{i+1}, \ldots, t_2^m)\}$ };
**if** $i \notin \mathcal{M}_g$ and $t_1^i \in \mathcal{U}'$ **then** $\mathcal{U} \leftarrow \mathcal{U} \cup \{t_1\}$ } }.

**(7)** [Direct embeddings in $f$.] **if** $|\mathcal{D}_f| = 1$ **then** {
let $\mathcal{D}_f = \{i\}$; $\mathcal{U}' \leftarrow \mathsf{CSMUBGT}(t_1, t_2^i)$;
**if** $\mathcal{U}' \neq \emptyset$ **then** { **if** $i \in \mathcal{M}_f$ **then** { **for** $t' \in \mathcal{U}'$ **do** $\mathcal{U} \leftarrow \mathcal{U} \cup \{f(t_2^1, \ldots, t_2^{i-1}, t', t_2^{i+1}, \ldots, t_2^n)\}$ };
**if** $i \notin \mathcal{M}_f$ and $t_2^i \in \mathcal{U}'$ **then** $\mathcal{U} \leftarrow \mathcal{U} \cup \{t_2\}$ } }.

Figure 8.13: An algorithm computing a complete set of minimal upper bounds

and does not have to perform a search on all type constructors, it is not necessary that the signature is finite. It is only necessary that there is an effective algorithm which computes for any type constructor $f$ the sets $\mathcal{D}_f$ and $\mathcal{M}_f$, and that the conditions imposed on algorithm CSMUBBT are fulfilled.[64]

[64] If the signature is finite, these conditions will always be fulfilled if the coercions between the base types are effectively given.

An example of an infinite signature with such properties is a finite signature extended with a type constructor $\mathtt{M}_{m,n}$ for any $m, n \in \mathbb{N}$ with the intended meaning of building the $m \times n$-matrices over commutative rings. It is natural to define $\mathcal{M}_{\mathtt{M}_{m,n}} = \{1\}$ for all $m, n \in \mathbb{N}$ and to have $\mathcal{D}_{\mathtt{M}_{m,n}} = \emptyset$ for $m \neq n$ and $\mathcal{D}_{\mathtt{M}_{n,n}} = \{1\}$ for any $n \in \mathbb{N}$.

#### Complexity of Type Inference

In [Wand89] and [Linc92] the complexity of type inference for expressions of the $\lambda$-calculus which are typed by allowing various possibilities of coercions are investigated.

In [Linc92] the problem is shown to be NP-hard if the order given by the coercions is arbitrary but fixed by reducing the following problem on partial orders called Pol-Sat to it.

> Given a partial order $\langle P, \leq \rangle$ and a set of inequalities $I$ of the form $p \leq w$, $w \leq w'$, where $w$ and $w'$ are variables, and $p$ is a constant drawn from $P$, is there an assignment from variables to members of $P$ that satisfies all inequalities of $I$?

Pol-Sat is an NP-complete problem. It is shown to be NP-hard by reducing the 3-SAT-problem to it.[65] However, if only lattices are allowed as partial orders in Pol-Sat then the problem is decidable in linear time.

A quite similar problem on partial orders, called Po-Sat is introduced in [Wand89], which is reduced in polynomial time to a type inference problem using polymorphic functions. The problem Po-Sat is proven to be NP-complete for arbitrary partial orders but to be solvable in polynomial time if the partial orders are restricted to finite quasi-lattices.

A quite systematic study of the complexity of decision problems for various partial orders which might be relevant for type inference is given in [Tiur92].

## 8.4 Other Typing Constructs

### 8.4.1 Partial Functions

Many functions arising in the area of computer algebra are only partially defined. Some basic examples are

1. division in a field, which is defined for non-zero elements only;
2. matrices over fields have inverses only, if they are regular;
3. the square-root over the reals exists for non-negative values only.

We could make partial functions total by introducing new types — the type of elements, on which the function is defined.

The following examples, which are taken from [Farm90], show that there are severe problems if we were to take this solution.

Let $f$ be the binary functions over the reals defined by

$$f(x, y) = \sqrt{x - y}.$$

The function $f$ cannot be represented as a *binary* total-function in a many-sorted algebra since the domain of $f$ is not a set of the form $D_x \times D_y$, where $D_x$ and $D_y$ are subsets of the

[65] A proof that 3-SAT is NP-complete can be found e. g. in [Davi94, p. 347].

real numbers.

It makes good sense to view division in a field as a partial function with the second argument having the type of the field. If in the case of the rationals we were to restrict the second argument to a type "non-zero rationals", we would have made this function total. However, this solution has a severe drawback. A term such as $1/(2-1)$ is no longer well-formed, since "$-$" is a function into the rationals and not into the non-zero rationals only.

The usual solution which is taken in connection with many-sorted and order-sorted algebras uses the "opposite" way.

New elements — "error elements" — are introduced and new types are built by adjoining these error elements. A partial function is made total by setting the value of the function to be an error element if it is undefined before, see e. g. [Smol89] for a more detailed description of this construction.

This construction is also used in universal algebra in order to embed a partial algebra in a full algebra, see e. g. [Grae79, p. 79].

In the area of computer algebra this approach is taken in the computer algebra system Axiom.

The disadvantage of this approach is that we loose information. If we consider terms built out of partial functions and total functions, we have to repeat the construction. Since the range of the partial function has increased, a previously total function has become partial, since it is not defined on the error value.

In the general framework of many-sorted or order-sorted computations, it might be difficult to regain the lost information. There are important examples, where the set of elements on which a partial function is defined is only recursively enumerable but not recursive (see e. g. [Smol89, p. 342] for an example).

In connection with a computer algebra system, a better solution should be possible. In most cases, the set of elements a partial function is defined on can easily be decided; in our examples a simple test for being non-zero, non-negative or calculating a determinant would have been sufficient.

Hence, in these cases it is decidable whether a *ground term* is well formed, i. e. has an error value or not.

Finding conditions and algorithms which tell the (minimal) type of an arbitrary term is an interesting problem, whose solution would be of practical significance.

#### Retractions

The sum of two polynomials is in general again a polynomial. However, if we add the polynomials $(-x+5)$ and $(x+2)$, we obtain the constant polynomial 3 as a result. For future computations it would be useful if we *retract* the type of the result from `integral polynomial` to `integer`

Since retractions are partially defined implicit conversion functions the general framework developed for other kinds of partial functions also applies to retractions.

### 8.4.2 Types Depending on Elements

In this section we will discuss typing constructs which correspond to the case of elements as parameters to domain constructors in Axiom. We will use the term "types depending on

elements" to describe these types, because it seems to be more or less standard for type theories including such constructs.

There are some important examples of data structures whose type depends on a non-negative integer.

- Elements of $\mathbb{Z}_m$.
- Vectors of dimension $n$.
- The $m \times n$-matrices.

However, the elements a type can depend on are not restricted to integers.

An algebraic number $\alpha$ over $\mathbb{Q}$ is usually represented by its minimal polynomial over the rationals. Thus, an element of the field $\mathbb{Q}[\alpha]$ has a type depending on some polynomial over the rationals.

An example of a type which depends on a matrix (namely the matrix defining a quadratic form) is the one which is built by the domain constructor `CliffordAlgebra` (see [Jenk92, Sec. 9.9].

In group theory programs, very often a group is represented with respect to its generators, cf. [GAPx17]. So the concept of types depending on elements is a possibility to treat certain structures which are treated as objects of a computation in a certain context as *types* in another one (cf. Sec. 8.2.6).

Some of the examples given above could be reformulated such that the concept of types depending on elements is no longer necessary in order to describe them. So it might be sufficient to have only a type of matrices of arbitrary dimension (over some ring) in the system and not a type of $m \times n$-matrices. Then matrix-multiplication or even addition of two matrices would be partial functions only. A treatment of partial functions (cf. Sec. 8.4.1) would be sufficient and the additional concept of types depending on elements could be avoided.

However, for the case of $\mathbb{Z}_m$ it seems to be necessary to have for any $m \in \mathbb{N}$ also a type corresponding to $\mathbb{Z}_m$ in a system which also allows the possibility to have computations on the integer $m$.

So the concept of types depending on elements is important for many computer algebra applications. Unfortunately, as we will show below it is not possible to have type-safe compile-time type-checking.

### Undecidability of Type Checking

**Lemma 8.** *Let $\mathcal{R}$ be the class of unary recursive functions. Then the following questions are undecidable:*

1. *For $f \in \mathcal{R}$, is $f(x) = n$ for some fixed $n \in \mathbb{N}$ and for all $x$?*
2. *For $f \in \mathcal{R}$, is $f(x)$ a prime number for all $x$?*
3. *For $f \in \mathcal{R}$, is $\gcd(f(x), n) = 1$ for some fixed $n \in \mathbb{N}$ and for all $x$?*

**Proof** All of the questions above are equal to determining the membership of $f$ in certain classes of partial recursive functions, which are all non-trivial. So the lemma is proved by applying Rice's Theorem (see e. g. [Odif92, p. 150]). □

Assume that the language is universal, i. e. every partial recursive function can be computed

in the language. Assume that there is a type corresponding to $\mathbb{N}$ present in the language and that indeed every unary recursive function can be represented in the system as one having type $\mathbb{N} \longrightarrow \mathbb{N}$. Moreover, assume that there is a type corresponding to $\mathbb{Z}_m$ for any $m \in \mathbb{N}$.

Let $n \in \mathbb{N}$ and let $f : \mathbb{N} \longrightarrow \mathbb{N}$ be a unary recursive function. By Lemma 8 it cannot be decided by a compiler, whether $\mathbb{Z}_{f(x)}$ and $\mathbb{Z}_n$ are equal. Thus having $a \in \mathbb{Z}_{f(x)}$ and $b \in \mathbb{Z}_n$ and having a polymorphic operation `op` with type

$$\forall t \,.\, t \times t \longrightarrow \texttt{Boolean}$$

like the check for equality it cannot be decided at compile time whether

$$\texttt{op}(a, b)$$

is well typed.

Determining whether $\mathbb{Z}_{f(x)}$ is a field, i. e. whether $f(x)$ is prime is also not possible at compile time. So it cannot be decided whether computations requiring that $\mathbb{Z}_{f(x)}$ is a field are legal.

Since it cannot be decided by the compiler whether $\gcd(f(x), n) = 1$ it is also impossible to decide whether the lifting connected with the Chinese remainder theorem can be applied to an element of $\mathbb{Z}_{f(x)}$ and to one of $\mathbb{Z}_n$ giving one of $\mathbb{Z}_{f(x)\cdot n}$.

In the following we will show that it is necessary to allow such run-time computations of elements a type depends on for many important applications in computer algebra.

### Necessity of Run-Time Computations of Elements Types Depend on

Frequently, computations in $\mathbb{Z}_m$[66] are done in the context of computer algebra because of the following observation:

If one wants to have the solution for a problem over the integers, then it is often possible to compute a $b \in \mathbb{N}$ (a "bound") such that for all $n \geq b$ the result of the computation in $\mathbb{Z}_n$ can easily be extended to a solution for the problem over the integers.[67]

Very often, these computations are not done directly in $\mathbb{Z}_b$, but in $\mathbb{Z}_{p_1}, \ldots, \mathbb{Z}_{p_h}$ for primes $p_1, \ldots, p_h$. The results are then "lifted" either to $\mathbb{Z}_{p_1 \cdots p_h}$ by an application of the Chinese remainder theorem or to $\mathbb{Z}_{p^l}$ by a Hensel lifting. The choice of $p_1, \ldots, p_h$ resp. of $p$ and $l$ are such that $p_1 \cdots p_h \geq b$ resp. $p^l \geq b$.

However, the class of algorithms which is used to compute the bounds can be fairly complicated. Technically speaking, if $f(x)$ and $g(x)$ are two functions that can be computed by the class of algorithms used for the bound computations, then it is undecidable whether

$$f(x) \equiv g(x) \qquad \forall x.$$

Let us now assume that we could restrict the occurring types to the ones corresponding to $\mathbb{Z}_{p_1 \cdots p_k}$, where $\{p_1, p_2, p_3, \ldots\}$ is the set of prime numbers. However, it is undecidable whether $p_1 \cdots p_k = p_1 \cdots p_{k'}$, if $k$ and $k'$ are minimal such that $p_1 \cdots p_k \geq f(x)$ and $p_1 \cdots p_{k'} \geq g(x)$. So a compiler cannot decide whether a statement involving an element of $\mathbb{Z}_{p_1 \cdots p_k}$ and one of $\mathbb{Z}_{p_1 \cdots p_{k'}}$ requiring both to have the same type[68] will lead to a typing error or not.

[66] Or in the ring of polynomials over $\mathbb{Z}_m$, etc. In our framework these structures can be all expressed as types having $\mathbb{Z}_m$ substituted for a type variable.

[67] Many books on computer algebra can serve as references, e. g. [Buch82] — especially [Laue82] or [Kalt83a] — or [Dave88], [Lips81], [Gedd92], and also [Knut71].

[68] Simple operations such as a test for equality or addition can serve as examples.

### Calculi Dealing with Types Depending on Elements

The results of this section show that it is useful to distinguish between domains and elements as parameters of domain constructors.

Having only type classes as additional typing construct a static typechecking is possible in principle in the former case. In the latter case it becomes undecidable, where we have argued that this undecidability results are relevant for many examples occurring in practical computer algebra applications.

For a user interface it is usually sufficient to perform type inference on expressions which do not allow recursion and which do not form a Turing-complete language for computations on elements types depend on.

So the problems which yield that the type inference problem and even the type checking problem is undecidable in the case of a computer algebra language do not apply to the case of a user interface of a computer algebra system.

Since the type of an element another type depends on can nevertheless be quite complicated (see the examples given above) it seems to be useful to have some sophisticated techniques available also for this case.

During the last years several general type theories having the concept "types depending on elements" have been developed. Some are Martin-Löf's Type Theory [Mart80], and the *Calculus of Constructions* of Coquand and Huet [Coqu86], They have been explored extensively, especially as "logical frameworks" [Huet91]. For this purpose several subcalculi and variations such as LF [Harp93], or Elf [Pfen89], [Pfen91], [Pfen92] have been defined. Some extensions of unification algorithms to these type theories have been given in [Elli89], [Pfen91a]. For the purpose of computer algebra probably another variant of this theories will be more suited than the existing. Nevertheless, it seems to be very likely that some of the obtained results are applicable to the type inference problem for a user interface of a computer algebra system.

# Chapter 9

# A Type System for Computer Algebra

This is based on Santas [Sant95]. Changes have been made to integrate it.

Type systems for computer algebra systems like Axiom [Jenk92] and theorem provers like Nuprl [Cons85], or functional programming languages like Standard ML (SML) [Miln90] [Miln91] [Harp93a], Haskell [Huda92], etc. and object oriented languages [Meye92] [Gold83] [Grog91] are usually stratified into levels or universes in a way similar to type theories like Damas and Milner's [Dama82] type schemes, Martin-Lof's [Mart73] constructive mathematics, etc.

More specifically, Axiom's types are basically divided into `domains` and `categories`; Domains are both *types* for run-time values like $Integer$, $Integer \rightarrow Boolean$ and $Fraction(Integer)$, and *packages* which include definitions of other values. Categories are *type* of domains, like $Ring$ and $Module(R)$. Categories and domains can be parameterized with other domains, allowing the definition of category and domain constructors. Categories specify the values defined in domains, and can form hierarchies of specifications.

SML's types belong to two universes: $U_1$, which includes `types` like $Integer$ and $String \rightarrow Boolean$, and the universe $U_2$ of types of `modules`; this system as extended by Mitchell, Meldal, and Madhav [Mitc91b] can include existential types ($\exists$ types) in $U_1$ and dependent product and sum types ($\Pi$ and $\Sigma$ types) in $U_2$. These two levels correspond essentially to the separation into `monomorphic` and `polymorphic` type expressions in the Damas and Milner [Dama82] system. *Monotypes* are arbitrary base types, an infinite number of `type variables` and the types built from other monomorphic types through function space constructor $\rightarrow$. *Polytypes* are the types which range over monotypes, through the universal quantifier $\forall$, like the type of the identity function: $\forall t.t \rightarrow t$. Since every monotype $\tau$ can be viewed as a polytype (like application of $\forall t.\tau$ to any monotype where $t$ is a type variable which does not exist in the type expression $\tau$), we assume that there is an injection from $U_1$ to $U_2$.

On the other side of the camp, object oriented languages include the notions of `class` and `abstract class`, resulting in (a) implementation classes which can define `objects`, and (b) abstract classes used explicitly for inheritance and specification of behaviour.

The two-universe approach in Axiom has been designed and proved beneficial for the mod-

eling of algebraic concepts, where *categories* correspond to *algebras* and *domains* to sets of values which share common representation and functionality as described by Davenport, Trager [Dave90] and Gianni [Dave91]. Similar results have been accomplished in object oriented languages like C++ [Stro95], where abstract classes play the role of categories, while implementation classes model domains in Barton and Nackman [Bart94]. On the other hand, the two universes and their corresponding elements in SML were design decisions for purely type-theoretic reasons. Although approaches vary, it is possible to define a mapping of the constructs defined in typed computer algebra systems to constructs of type theory similar to the ones used for the definition of programming languages like SML. This is important in two ways: it clarifies the semantics of computer algebra systems, and allows their further improvement by means of extensions, removal of inconsistencies, more expressiveness, and better performance. We try to accomplish these targets with the definition of a strong and static type system based on a powerful subset of the Axiom system, which we extend incrementally.

We briefly examine some of the most recent concepts for object oriented programming and abstract datatypes for computer algebra systems. We observe that these concepts although quite powerful are not adequate for the properly typed modeling of the relations among simple algebraic constructs like the domains of integers and rationals. These concepts are simplified and integrated with the notions of domains and categories. We provide the formal definition of domain categories by means of existential types and the calculus for subtyping on categories. Our type system is enriched with transparent types which form the main construct of the sharing mechanism. Domains are extended to form packages while their corresponding types form dependent sums. Parameterized categories and functors are examined and their calculus is defined with subtyping rules. Sharing constraints are proved to handle many cases of parameterization resulting in flattening of parameterized categories with facilities for recursive definitions at the category level; there we establish the conditions for successful functor application. This reduces the load of the type system considerably, so as to afford the inference and manipulation of type classes.

Since Axiom has the most references in the literature of typed computer algebra systems, the notation used here is based on an Axiom-like language which assumes:

1. values which consist of value variables, records, record components, (higher order) functions, term application, value components of packages and *let* expressions
2. type expressions with type variables, base types, record types, tagged unions, recursive types, function types, type components from packages, type constructors, and type application
3. domain expressions with domain variables, self, domain constructors, parameterized domains and application
4. expressions which can be used either as domains or types
5. packages with package variables, package constructors, package components of other packages, parameterized packages and application
6. categories with category variables, sharing, category constructors, parameterized categories and application
7. sharing constraints
8. facilities for accessing package components

The abstract syntax of the language is given by the following grammar:

| | |
|---|---|
| values : | $e ::= x \mid \{x_i = e\}_{\forall i \in I} \mid e.x \mid (x : \delta) \mapsto e \mid e\ e \mid x\$w \mid let\ x : \phi := e\ in\ e$ |
| types : | $\tau ::= t \mid T \mid \{x_i : \tau\}_{\forall i \in I} \mid +[x_i : \tau]_{\forall i \in I} \mid \mu t.\tau \mid t\$\omega \mid \tau \to \tau \mid t \mapsto \tau \mid \tau\ \tau$ |
| domains : | $\delta ::= d \mid \% \mid add(\overline{\delta}) \mid (d : \sigma) \mapsto \delta \mid \delta\ \omega$ |
| both : | $\phi ::= \delta \mid \tau$ |
| packages : | $\pi ::= m \mid add(\overline{\pi}) \mid m\$\omega \mid (m : \sigma) \mapsto \pi \mid \pi\ \pi$ |
| categories : | $\sigma ::= c \mid with(\overline{\sigma}) \mid (m : \sigma) \mapsto \sigma \mid c\ \omega$ |
| $\delta -$ comps : | $\overline{\delta} ::= \emptyset \mid Rep := \phi; \overline{\delta} \mid x := e; \overline{\delta} \mid \delta; \overline{\delta}$ |
| $\pi -$ comps : | $\overline{\pi} ::= \overline{\delta} \mid m := \pi;\ \overline{\pi} \mid t := r;\ \overline{\pi} \mid \pi;\ \overline{\pi}$ |
| $\sigma -$ comps : | $\overline{\sigma} ::= \emptyset \mid m : \sigma;\ \overline{\sigma} \mid t : Type;\ \overline{\sigma} \mid s;\ \overline{\sigma} \mid x : \tau;\ \overline{\sigma} \mid \sigma;\ \overline{\sigma}$ |
| sharings : | $s ::= t = \tau \mid t = d \mid t = d\$w \mid m = \omega$ |
| access : | $\omega ::= m \mid \omega\ \omega \mid m\$\omega$ |

The components inside the *with* and *add* constructs are optional. *let* expressions can be defined for types, domains, packages, and categories. Declarations in value expressions are optional, thus $let\ x := e\ in\ e$ is valid too. Wherever we have $t = thing$ or $t : Type$ we can assume also that $Rep = thing$ and $Rep : type$ is allowed, but not vice versa. Records with $n > 0$ components are represented as $\{x_i : \tau\}_{\forall i \in \{1,\ldots,n\}}$. Similarly for unions, replacing $\{\}$ with $+[]$. Values can include $n$-tuples of the form $(e, \ldots, e)$ which are implicitly viewed as records indexed by integers from 1 to $n$; the expression $()$ is the empty tuple and is the same as $\{\}$; there are no 1-tuples. Tagged unions can model enumerated types or variations as in:

$$bool := true = false$$

$$tree(t : Type) := empty + node : (tree\ t, tree\ t) + leaf : t$$

The syntax of this language hardly distinguishes between domains and packages; this comes in accordance with the Axiom language where domains are packages. Since domains can be used in a context where types are expected, $\phi$ defines a dummy union of domains and types. Types can be lifted to domains, and the representation of domains can be viewed as type. The exact transformations are examined in what follows; this allows us to have domains in union and record types without complicating the language. Additionally packages assume the arrow types as special cases of $\Pi$ types where there are no dependencies between the domain and the range of the types.

The calculus of the presented type system is given by a set of selected rules. Each rule has a number of antecedent judgments above a horizontal line (optional) and a conclusion judgment below the line. Each judgment has the form $E \vdash \mathcal{A}$, for an environment $E$ and an assertion $\mathcal{A}$, depending on the judgment. Environments contain typing assumptions for variables, type variable declarations and subtyping assumptions. Typical judgments are $E \vdash x$ asserting that $x$ is definable in $E$, $E \vdash x : \sigma$ meaning that $x$ has type $\sigma$ under assumptions $E$ and $E \vdash \sigma' \sqsubseteq \sigma$ asserting that $\sigma'$ is a subtype of $\sigma$ (or equivalently, $\sigma$ is a supertype of $\sigma'$) in $E$; we demand that all the environments be well formed. An environment can be extended with new assumptions as in $E; x : \sigma; E' \vdash x : \sigma$; this environment is well formed provided that there is no other assumption for $x$ in $E; E'$; variables can be renamed in case of conflicts.

Some of the examples given have been influenced by the modeling of algebraic concepts in the Axiom library.

Although demonstration of the simplicity of our type system is out of the scope of this chapter, we claim that it combines the concepts of `domain` and `package` and Haskell-like `type classes`, it increases the expressiveness of the programs with the propagation of sharing con-

straints, it distinguishes compile time and run-time constructs allowing for static and strong type checking and more optimizations and reduces the need for `coercions` and `retractions` (described in Sutor and Jenks [Suto87] and analyzed by Fortenbacher [Fort90] in Axiom.

### 9.0.3 Object in Computer Algebra

Many computer algebra operations involve purely functional objects: functions or methods operate on objects and return a new object with a new internal state, instead of updating the state of an older object. If the objects are big (like lists, arrays, or polynomials) then in-place updating is more common. In object-oriented programming however there is always an updating of the internal state of an object: an `object` must be a `reference`. Due to the poor type-theoretic properties of references and side-effects, most type theoretic accounts of object-oriented programming by Cardelli and Wegner [Card85], Cardelli [Card86], Ghelli [Ghel91], Pierce [Pier91], etc. deal with purely functional objects and employ techniques used for purely functional closures, or define special constructors for dealing with the increased complexity of object subsumption and effects.

Object oriented languages usually take one object (e.g. a rational number) with methods and functions for reading and updating its numerator and denominator or performing addition among rationals - to be an element of a recursively defined type (Bruce and Mitchell [Bruc92]) like:

```
Rational : Category := with (
  new: (Integer,Integer) -> Rep ;
  numerator: Rep -> Integer ;
  denominator: Rep -> Integer ;
  SetNumDenom: Rep - (Integer,Integer) -> Rep ;
  + : Rep -> Rep -> Rep ;
  * : Rep -> Rep -> Rep ; etc. )
```

where *Rep* stands for the type under scope, here *Rational*, or any subtype of *Rational*.

This encoding hides the fact that Rationals may have an internal state that is shared by all its methods: the responsibility for building a new instance of Rational in response to a call to *new* is placed within the function *new* itself. Another notation identifies the representation of instances with the implementation of the representation, as in Pierce and Turner [Pier93]

```
RationalFun := (Rep: Type) -> with (
  new: (Integer,Integer) -> Rep ;
  numerator: Rep -> Integer ;
  denominator: Rep -> Integer ;
  SetNumDenom: Rep -> (Integer,Integer) -> Rep ;
  + : Rep -> Rep -> Rep ;
  * : Rep -> Rep -> Rep, etc. )
```

This method gives the caller of *new* the responsibility for transforming the value returned by *new* into a new Rational instance. Although this notation may seem strange, it offers much in terms of *simplicity*: the entire development, including the notions of *Rep*, can be carried out without the need of `recursive types`, dealt by Amadio and Cardelli [Amad93] or `extensible records` (Cardelli [Card88a], Mitchell [Mitc88a], Wirth [Wirt88]) which are implementation dependent; however, expressiveness is sacrificed. Using recursive types the above expressions can have any desirable formulation as in

```
RationalFun := (Rep < F(Rep)) -> with (
  new: (Integer,Integer) -> Rep ;
```

```
  etc. )
```

where **F** can be the functor *RationalFun* itself. This formulation is closer to our intuition about object-oriented programming with subtypes. It can be further enriched with other second-order constructs, which lead to the definition of special `object constructors` by Abadi and Cardelli ([Abad94] [Abad94a]).

These scheme are based on `existential types` (Mitchell and Plotkin [Mitc88]) which hide the implementation of the objects, allowing us to deal with first class `abstract types` (Liskov [Lisk79], Burstall and Lampson [Lamp88], Milner and Tofte [Miln91], Bruce and Mitchell [Bruc92]) independently of implementation details without any cost in type safety. In the next sections we present some examples of the advantages and the limitations of this approach: this leads us to the introduction of constructs which make the representations visible in order to add more flexibility to the type system keeping it consistent and sound while increasing the efficiency of the implementations.

*RationalFun* is actually the type of a `functor`, i.e. a *function from types to types* and specifies the visible behaviour of the functions on rationals, in terms of the *abstract representation type*. An object satisfying this specification consists of a list of functions of type *RationalFun(Rep)* for some concrete type *Rep*, paired with a state of type *Rep*: both are surrounded with an abstraction barrier that protects the `internal structure` from access except through the above specified functions. By internal structure we mean either the internal state, or the hidden functions that operate on this state, or any other operation including the ones mentioned in the specification. This encapsulation is directly expressed by a (possibly recursively defined) existential type:

```
Rational:=((Rep: Type) +> with (Rep, RationalFun(Rep))) SomeRep
```

Abstracting *RationalFun* from *Rational* yields a higher order type operator that, given a specification, forms the `type` of objects that satisfy it.

```
DeclareDomain := (Fun: Type -> Category) +>
  (((Rep: Type) +> with(Rep,Fun(Rep))) SomeRep)
```

The type of *Rational* objects can now be expressed by applying the *DeclareDomain* constructor to the specification of *RationalFun*:

```
Rational := DeclareDomain(RationalFun)
```

or the shortcut:

```
Rational : RationalFun
```

In order to give proper treatment to the interaction between representations and subtyping, it is necessary to separate *Rational* into the specifications of its functions and the operators which capture the common structure of all object types. This separation is also important for the semantical construction of categories and the definition of the internal structures of the types.

### 9.0.4 Multiple Representations

Rationals are created using the function *box*, which captures the semantics of dynamic objects in object oriented programming. A rational number with `representation` (x:Integer,y:Integer), `internal state` (5,2) and method implementations:

```
numerator := (r:{x:Integer, y:Integer}) +> r.x
denominator := (r:{x:Integer, y:Integer}) +> r.y
```

can be created as:

```
r: := box ( coerceTo ( ( Rep := {x:=5, y:=2},
  (numerator := (r:{x:Integer,y:Integer}) +> r.x ;
   + := (r:{x:Integer,y:Integer}) +> (arg: Rational) +>
     new((r.x*denominator arg + r.y*numerator arg)/
                              (r.y * denominator arg)
   etc. )),
  Rational ))
```

The *coerce* function here is only a syntactic construct, which shows the compiler how to view the introduced list; in this example it is the identity function since we provide exactly what is necessary for building a rational.

The *box* function is helpful for the implementation of the *new* function and the $+$ and $*$ operators which return new instances of Rational:

```
new:=(initX:Integer, initY:Integer) +>
   box ( coerceTo ( (Rep:={x:=initX,y:=initY},...), Rational))
```

Unlike Axiom, in object oriented systems based on `delegation` (Ungar and Smith [Unga91]) or on `subtyping` the elements of an object type have different internal representations and different internal representations of their functions. For example, a rational with representation type $(x : Integer)$ might be implemented as follows:

```
r2 := box ( coerceTo ( (Rep := {x:=5},
  (numerator := (r:{x:Integer}) +> r.x ;
   denominator := (r:{x:Integer}) +> 1 ;
   + := (r:{x:Integer}) +> (arg:Rational) +> new(r.x + numerator arg) ;
   etc. )),
  Rational ))
```

and the definition of *new* changes accordingly. The functions *SetNumDenom, numerator, denominator* have compatible signatures as in the previous implementation since we abstract $s : \{x : Integer\}$ to $s : Rep$.

Static type checking can be obtained even without having all the information about the internal structure of the objects. The constants 0 and 1 can be implemented in any of the two ways, having $x$ assigned to 0 and 1 respectively, while the $y$ field can be 1. This variability will be helpful for defining type classes in a later section.

Type checking however is complicated by the evaluation process. Suppose that we invoke the function *SetNumDenom* passing to it the object $r_2$. We have to open $r_2$, and make sure that the only functions applicable to its *Rep* are the ones declared in *Rational*. Consequently we produce a new value of type *Rep*, which is *reboxed* as a rational number which has access to the methods of $r_2$, and hides *Rep*. The same process happens for functions like $+$ and $*$. The functions *numerator* and *denominator* do not return any new object, so no reboxing is necessary. The complications appear when `binary operations` are introduced in this schema. Since the two arguments of the *plus* operator may have different representations, it is *necessary* that the objects are accessed through the functions that operate on their structure, if the compiler allows this at all. However, calling *plus* from $r_1$ does not necessarily return the same results as *plus* from $r_2$. The problem can be transferred one level deeper, namely at binary operations on integers, which might pose similar representation complications. This is a result of the pure object oriented scenario with message passing we have assumed so far. Although such asymmetries do not seem appealing for ordinary arithmetic operations, the above constructs are quite useful for the implementation of operations which depend

on one argument or operations on elements of heterogeneous lists. In order to handle the above, some object oriented systems are extended with generic functions: this results in very complex calculi for modeling their behaviour, which do not scale up easily for applications in computer algebra.

### 9.0.5 Domains and Categories

It is useful to have all the operations defined in the previous section in the class *Rational*. In such case we assume a hidden *common representation* for all objects of class *Rational*, so we can assume that Rational is defined locally:

```
let Rep := {x:Integer, y:Integer}
in RationalDomain := coerce (
  (new := (m:Integer, n:Integer) +> box {x:=m, y:=n} ;
   numerator := (s:Rep) +> s.x ;
   + := (s1:Rep) +> (s2:Rep) +> box(...) ;
   etc. ),
  RationalFun Rep)
```

Many simplifications can be performed. Since all the objects of *RationalDomain* have the same representation, unboxing and reboxing is not needed for performing the operations *SetNumDenom*, +, etc. These functions can operate directly on the representations of the objects. By doing this we reduce the computational cost of the boxing operations, be we restrict the application of the above functions to objects of the same domain. The function *coerce* replaces syntactically all the annotations which include *Rep* with annotations which include *RationalDomain*. There are two main reasons for this: since *Rep* has been defined locally, it should not escape this local scope; additionally, we may want *Rep* to be opaque, so that other definitions do not depend on the implementation of this particular representation.

The above construct is not unique in Axiom. It corresponds to `abstype` declarations in SML, or `clusters` introduced by Liskov, Snyder, Atkinson, and Schaffert [Lisk77] [Lisk79] in CLU and, as we saw before, classes with `private members` in object oriented languages.

#### Categories

We have seen that the instances of a class may have various representations, while it is desirable to have an abstraction on the level of functions defined for the instances of a class. Mitchell and Plotkin [Mitc88] proposed that the *interface part of a data type be viewed as a type, while the implementation part as a value of that type*. According to this analysis a domain comprises:

1. the internal representation of its instances
2. operations for analyzing and manipulating this representation
3. an abstraction barrier that prevents any access to the representation except by means of the given operations

Some of the functions defined in the example with rationals, like *numerator*, *denominator*, *SetNumDenom*, and *new* deal only with the representation of its instances. These functions form the *type of the representation*. Other functions like $+$, $*$, $/$, and the constants 0 and 1 are expected to be declared in a transparent fashion to any form of implementation. The second set of operations can be included in the specification of many other types, independently of the operations which manipulate the representation. For instance $(+)$ can be defined for

Matrices and Booleans, without any need for them to include operations like *numerator* or *denominator*. At the same time, a (+) defined to operate on rationals, should not accept matrix values as arguments.

We can transliterate this, using `categories`:

```
RationalMeta := (Rep:Type) +> (
  + : Rep -> Rep -> Rep ;
  - : Rep -> Rep ;
  * : Rep -> Rep -> Rep ;
  / : Rep -> Rep -> Rep ;
  0 : Rep ;
  1 : Rep ;
```

This specification can be viewed as a coding for the `algebraic structure` *Field*: we write

```
Field:=RationalMeta
```

*Rep* refers to the implementation of the domains declared as instances of Field:

```
Rational : Field
```

The programming language *Haskell* defines similar constructs calling them `typeclasses`. The analogous constructs for object oriented languages are `abstract classes`, although they cannot express the constraint we have posed above.

Axiom's categories and object oriented abstract classes can provide `default definitions` in categories which can be overridden in other categories or in the domains instances of these categories. This construct has great expressive capabilities but we do not handle it in this paper, because its semantics are not well understood and may cause unsoundness in the type system. We will examine an alternative, which can be extended to support default definitions at category level.

### Existential Types

Categories are the way `existential types` are coded in the type system. The category *Field* corresponds to the existential type [1].

$$\exists Rep \quad [(Rep, Rep) \to Rep, Rep \to Rep, \\ (Rep, Rep) \to Rep, (Rep, Rep) \to Rep, Rep, Rep]$$

This means that there exists a type *Rep*, which is used for the implementation of the operations *plus*, *minus*, etc. with the types $(Rep, Rep) \to Rep$, $Rep \to Rep$, etc. respectively.

Existential types ([Mitc88],[Card85]) were introduced in constructive logic by Girard [Gira72] and are related to `infinite sums` in *category theory* (Herrlich and Strecker [Herr73]). If all the domains have a hidden representation, they are all described by types of the form: $\exists Rep.C$ where *Rep* is free in $C$. In the domain declaration

$$D : C := add(Rep := \tau, M)$$

the type of the components of $M$ is the result of substituting $\tau$ for *Rep* in $M_i$ doing the appropriate $\alpha$-conversions if needed:

$$M_i : [\tau/Rep]C_i$$

[1] The fact that the name of the operators is lost in the translation is of minor importance. We can assume canonical representation of the components of *Field* corresponding to the ordering in the resulting list

$$[\text{sub}-\text{refl}] \qquad E \vdash M \sqsubseteq M$$

$$[\text{sub}-\text{trans}] \qquad \frac{E \vdash S \sqsubseteq T \quad E \vdash T \sqsubseteq U}{E \vdash S \sqsubseteq U}$$

$$[\text{thinning}] \qquad \frac{E \vdash M_a \sqsubseteq M_b}{E \vdash [\overline{M_a}, M_{n+1}] \sqsubseteq [\overline{M_b}]} \quad \overline{\{M_1, \ldots, M_n\}} = M_1, \ldots, M_n$$

$$[\exists-\text{opaque}] \qquad \frac{E, Rep : Type \vdash M_a \sqsubseteq M_b}{E \vdash \exists Rep.M_a \sqsubseteq \exists Rep.M_b}$$

$$[\exists-\text{transparent}] \qquad \frac{E, Rep = \tau \vdash M_1 \sqsubseteq M_2}{E \vdash \exists Rep = \tau.M_1 \sqsubseteq \exists Rep = \tau.M_2}$$

$$[\exists-\text{forget}] \qquad \frac{E, Rep = \tau \vdash M_1 \sqsubseteq M_2}{E \vdash \exists Rep = \tau.M_1 \sqsubseteq \exists Rep.M_2}$$

Figure 9.1: Subtyping rules for ∃-types

allowing the type-checking of the definition of a domain. If a domain does not define any *Rep*, then a `type variable` is assumed for the typing process; thus, *Rep* can be instantiated to any other type in extensions of this domain.

By declaring a variable as an instance of a domain $D$, the following constants are introduced:

$$f_i : [D/Rep]C_i$$

i.e. the name of the domain $D$ replaces all occurrences of *Rep* in the above context. The only information provided to the global context about $D$ is its name and its exported operations. If its representation is visible the type system can infer the $D = \tau$.

Since representations may involve recursive definitions as in the case of lists, we introduce the notion of `recursive types`. A recursive type $\mu t.\tau$ satisfies the equation:

$$\mu t.\tau = [\mu t.\tau/t]\tau$$

In order to avoid the continuous replacement of *Rep* in $D$ in a recursively defined domain, we can perform the substitution only once, and then bind the nested occurrence of *Rep* to a fresh variable which can be equated to $D$ when needed during the inference process. In this way, we abstract recursive types away, by using one level of indirection provided by the existential types. The penalty we pay is that type equality for recursive types cannot be structural. The same type definition in two different contexts will introduce two different fresh variables and consequently two different types. Although the expressiveness is not the same as in the case of pure recursive types, since we cannot reason about type equality as in (Amadio and Cardelli [Amad93]), this notation can model all the cases of recursive types. [Mitc88].

Domains have the `transparent type`:

$$\exists Rep = \tau.C$$

where *Rep* is free in $C$. This corresponds to a category where the representation of a domain is visible.

The transparent type $\exists Rep = \tau.C$ can be reduced to the `opaque type` $\exists Rep.C$ by *forgetting or hiding* the constraint $Rep = \tau$. In such case all the information about the representation of $D$ is lost: $D$ is an `abstract type`.

### Subtyping among Categories

Categories build hierarchies analogous to the existing hierarchies in algebra as described by Davenport and Trager [Dave90]. Although algebraic hierarchies are based on set theoretic terms concerning the properties of the algebraic structures, in a type system we can assume only type theoretic terms, which, in our case, correspond to algebraic constructs.

Intuitively a category $B$ is a subtype of category $A$ (we write this as: $B \sqsubseteq A$), if each domain belonging to $B$ belongs to $A$, or satisfies the properties of $A$. The subtype relation is reflexive and transitive. The rules for subtyping are given in Figure 9.1.

A Category $B$ extends category $A$ if its definition introduces new components. In such case, $B$ is a subtype of $A$ [thinning]. The subtype relation [$\exists$-opaque] between two opaque categories reduces to the previous rule. The [$\exists$-transparent] rule for transparent categories is slightly more complex: it requires that the representations of their instances are the same, and then reduces to the thinning rule. The rule [$\exists$-forget] involves forgetting the information about the domain representations. For a category with a constraint on the representation of its domains to be a subtype of a category without this constraint, we check if the former can form a subtype of the latter when the constraint is added. A category with fields labeled $with(x_1 : \sigma_1, \ldots, x_m : \sigma_m)$ is a subtype of any category with a smaller collection of fields, where the type of each shared field in the *less informative* type is a supertype of the one in the *more informative* type.

The above scheme is more general than Axiom's. Axiom allows subtyping among named categories only if this has been explicitly defined by the user. However the introduction of `anonymous categories` in Axiom as described by Jenks and Sutor [Jenk92], should involve rules like the above modulo the transparent types (the current version forbids the declaration of transparent categories).

Another advantage of the above scheme is that representation constraints can be specified directly in the categories: each domain whose type is $\exists Rep = \tau.M$ should define a representation equal to $\tau$. The reason we allow this is to make the system more flexible while maintaining consistency, so that `domain extensions` do not lead to unsoundness.

The last rule is a special case of the rule for subtyping defined in a new type calculus for SML by Harper and Lillibridge [Harp94]: $\tau$ expressions cannot include $\exists$-types in our scheme, separating the universe of categories from the universe of types. They can be shared with no anonymous domains though. Without this stratification, using the above general version would result in undecidability, since the substitution $[\tau/Rep]M_2$ can increase the size of type expressions as shown by Pierce [Pier91a]. Additionally, sharing with domains from given paths gives a meaning to domain equality which would be otherwise very difficult to obtain. In this way we can always decide if two domains share by checking their paths: if two domains derive from the same path then they share, otherwise we have to examine the sharing constraints in which they participate.

### 9.0.6 Domain Sharing

In the previous section we mentioned the presence of `transparent types`. The difference between opaque types, is that the information about the representation of a domain is visible in the case of transparency, and invisible otherwise. This means that in the declaration

$$D : \exists Rep = Integer.[f : Rep \to Rep, x : Rep]$$

the domain $D$ shares the same representation as the domain $Integer$, and the value components $x$ can be safely viewed as $Integer$. This allows a certain level of flexibility in the manipulation of domains and their components.

Sharings introduces a form of equational reasoning in the system, similar to that presented by Martin-Löf in his intuitionistic type theory [Mart73]. The behaviour of sharings is summarized in the following rule for domains with transparent types:

$$[\exists-\text{sharing}] \quad \frac{E \vdash D : \exists Rep = \tau.M}{E \vdash D = \tau}$$

This rule comes in accordance with Axiom's behaviour where the representation of a domain can be exchanged with the domain itself inside the scope of its definition.

The first application of this construct is the extension of domains. In the above example, the domain $D$ extends the domain $Integer$ by introducing the function $F : Integer \to Integer$ and the integer value $x : Integer$. Every new domain which shares representation with $D$ and $Integer$ is again an extension of any of them.

Opaque and transparent types can be combined to control domain extensions and ensure their safety. In the definitions

```
Semigroup := with (+ : Rep -> Rep -> Rep )
Monoid := with ( Semigroup; 0:Rep )
D1 : with (Semigroup; Rep=Integer) := add (
   Rep := Integer ;
   + := (+)$Integer )
D2 : Monoid := add (
   D1 ;
   Rep := D1 ;
   0:Rep := 0$Integer )
```

the domain $D_1$ shares with $Integer$. This allows a safe extension of $D_1$ in the body of $D_2$, which involves a value of type $Integer$. Since the type $Monoid$ of $D_2$ does not specify any sharing, $D_2$ is sharing with neither $Integer$ nor $D_1$. This does not mean that $D_2$ cannot be extended:

```
D3 : with ( Monoid; Rep=D2 ; double: Rep -> Rep) := add ( D2;
            double(x:Rep):Rep := x+x )
```

$D_3$ shares with $D_2$; instances of $D_2$ can be passed as arguments to the function *double* defined in $D_3$.

A side-effect of domain sharing is that if two domains $D_1$ and $D_2$ share common representation and define functions with the same name $f$ such that

$$f_{D_1} : t_1, \quad f_{D_2} : t_2, \quad t_1 = [D_1/D_2]t2$$

then if both domains are imported in the same context, the system is unable to choose the right function; explicit selections (as we did above) are needed. This is part of a more general

problem which suggests that name overloading in the same context is not recommended, although identifiers should be free to have different meanings in different contexts.

### 9.0.7 Packages and Categories

Type systems which provide domains and abstract types can be transformed into systems with `module types`, by flattening domains into packages. Languages like *Modula-2* [Wirt83], *Ada* [Adax12], and *CAML* [Lero17] which do not use domains or classes for the implementation of objects use the notion of `package` or `module` for encapsulating the appropriate behaviour: each domain $D$ with representation $t$ and operations $M_i$, where $t$ is free in $M$, can be viewed as a package with a type component $t$ and value components $M_i$ which depend on the type $t$.

In such case it is useful to consider packages with type components as instances of categories. For example, a package which implements rationals, and its corresponding type are:

```
RationalPack := (
  Rep := {x:Integer, y:Integer} ;
  + : Rep -> Rep -> Rep := a +> b +>
    {x := a.x*b.y + a.y*b.x, y:=a.x*b.x} ;
  ... )
```

```
FieldPack := with (
  Rep: Type ;
  + : Rep -> Rep -> Rep ;
  - : Rep -> Rep ;
  * : Rep -> Rep -> Rep ;
  ... )
```

```
RationalCatPack := with (FieldPack; Rep = {x:Integer, y:Integer})
```

Categories which correspond to packages represent a type similar to an existential type, namely, a `dependent sum type` (MacQueen [Macq84], Harper, Mitchell, and Moggi [Harp89], Constable et al [**?**]). Since all the type and value components of a package can be accessed, package categories are `strong sums` (MacQueen [Macq86]) of the form

$$\Sigma t : Type.[(t,t) \to t, \quad t \to t; ...]$$

For reasons of symmetry, domain categories whose type component cannot be accessed form *weak sums* (Harper and Lillibridge [Harp94]). This approach has been influenced by Martin-Löf's Constructive Mathematics [Mart79]).

Packages can be nested inside other packages. `Nested packages` are accessed by accessing their parent package recursively. The components of a package can depend on the type components of other nested packages as in:

```
PolynomialCat := with (
  R:Ring;
  E:OrderedAbelianMonoid;
  Rep:Type;
  leadingCoefficient: Rep -> Rep$R;
  degree: Rep -> Rep$E;
  ... )
```

In this way, *extensions* on existing domains are easily implemented, by flattening domains to packages, and exposing their representations which are matched with the representation

of the extensions. *Sharing* among type components are introduced locally inside a package, allowing the definitions of constraints which are valid only inside a certain scope and are invisible outside of it. This information is not easily expressible with domains without equating the two domains as we saw in a previous section.

Multiple type components are represented by `nested Σ types`, resulting in structures which contain two or more carrier sets (as in *PolynomialCat*). This implies that many complex data structures can be defined in packages and make their representations visible to the world.

The basic rules for subtyping on sum types are similar to the ones described for existential types but more general since multiple nested dependencies can occur in the body of a category. The rules are given in Figure 9.2. The rule Σ-instance needs special attention. We will see that types and values can be coerced to packages. Thus, type sharings are propagated together through the typing mechanism. In the absence of this mechanism, we would need to add additional rules such as

$$[\Sigma\text{−instance} - \tau] \quad \frac{E \vdash \tau \quad E; t = \tau \vdash \overline{\delta} : \overline{\sigma}}{E \vdash add(t = \tau, \overline{\delta}) : with(t : Type, \overline{\sigma})}$$

The correspondence between ∃-types and Σ-types is quite interesting and shows the set-theoretic distinction between the two constructs (Turner [Turn91, chapter 10], Mitchell and Plotkin [Mitc88]):

$$x \in \Sigma T : C.M \leftrightarrow \exists T : C.(x \in M)$$

This relation is of practical interest too, since it denotes that abstract types hide their implementations and components, but packages expose it. For instance, when a package with an opaque type $\Sigma t.M$ is opened it introduces the constants $x_i : M_i$ and $t : Type$, which is similar to those of domains. On the other hand, if a transparent package of the form $\Sigma t = \tau.M$ is opened, the introduced components are: $x_i : [\tau/t]M_i$ and the type $t = \tau$ whose kind shows that it shares the same implementation as $\tau$, i.e. they are the same types. This however does not mean that the two packages are the same, allowing us to make a distinction between type and package sharings (something we could not do with domains alone).

One could argue that packages with multiple type components correspond to multiple domain definitions. This is in general impossible when the distinction in the interface (category) for the two domains is not clear:

```
PackMultCat := with (
  R: Type ;
  S: Type ;
  x: R -> S -> R )
```

In order to implement the above package as a combination of two domain categories, each category needs to reference the representation of the domains of the other. This involves parameterization, which we discuss in the next section. In general Σ-types allow the typing of more expressions than ∃-types, since they introduce `impredicative polymorphism` (Harper and Mitchell [Harp93a]).

### 9.0.8 Parameterization

In the presentation of categories and domains in previous sections, we assumed that they are parameterized by the representation of their instances. This was merely a notational

$$[\Sigma\text{−inherit}] \quad \frac{E \vdash \sigma \sqsubseteq \sigma'}{E \vdash with(\sigma) \sqsubseteq with(\sigma')}$$

$$[\Sigma\text{−thinning}] \quad \frac{\forall i \in I. \exists j \in J. \quad E; (; [\overline{\sigma_j}])_{\forall j \in J} \vdash \overline{\sigma_j} \sqsubseteq \overline{\sigma'_i}}{E \vdash with(\overline{\sigma_j})_{\forall j \in J} \sqsubseteq with(\overline{\sigma'_i})_{\forall i \in I}}$$

$$[\Sigma\text{−sub−val}] \quad \frac{E \vdash \tau' \sqsubseteq \tau}{E \vdash x : \tau' \sqsubseteq x : \tau}$$

$$[\Sigma\text{−transparent}] \quad \frac{E \vdash \tau' = \tau}{E \vdash t = \tau' \sqsubseteq t = \tau} \qquad \frac{E \vdash \omega_1 = \omega_2}{E \vdash m = \omega_1 \sqsubseteq m = \omega_2}$$

$$[\Sigma\text{−forget}] \quad \frac{E \vdash \tau}{E \vdash t = \tau \sqsubseteq t}$$

$$[\Sigma\text{−opaque}] \quad E \vdash t \sqsubseteq t$$

$$[\Sigma\text{−decl}] \quad \frac{E \vdash \sigma' \sqsubseteq \sigma}{E \vdash m : \sigma' \sqsubseteq m : \sigma}$$

$$[\Sigma\text{−nesting}] \quad \frac{E \vdash \sigma' \sqsubseteq \sigma \quad E, m : \sigma' \vdash \overline{\sigma'} \sqsubseteq \overline{\sigma}}{E \vdash with(m : \sigma', \overline{\sigma'}) \sqsubseteq with(m : \sigma, \overline{\sigma})}$$

$$[\Sigma\text{−sub}] \quad \frac{E \vdash m : \sigma' \quad E \vdash \sigma' \sqsubseteq \sigma}{E \vdash m : \sigma}$$

$$[\Sigma\text{−instance}] \quad \frac{E \vdash \delta : \sigma \quad E, m : \sigma \vdash \overline{\delta} : \overline{\sigma}}{E \vdash add(m = \delta, \overline{\delta}) : with(m : \sigma, \overline{\sigma})}$$

Figure 9.2: Subtyping rules for ∃-types

convenience in order to clarify the correspondence between domains and their existential types. Computer algebra systems like Axiom have chosen to use a special symbol such as $ or % for representing the unaccessible type component of the domain categories. We used a *flattened* notation for packages in order to specify explicitly that type components can be accessible as if they were normal value components. Nevertheless, the code previously given for the creation of the objects $r_1$ and $r_2$, provides a concrete type for the representation as it is difficult to implement the functions without knowledge of the actual hidden type.

### Parameterized Categories

What is hidden behind the notation we used is that it is possible to parameterize both domains and packages with other domains and packages. Given the relation between ∃-types and Σ-types and the conclusion that packages are more general than domains, we can assume that Σ-types and packages can be parameterized by other packages resulting in new Σ-types and packages respectively as in:

```
DictionaryCat (Keys: with
                      (T:Type;
                       eq:T->T->Boolean;
                       less:T->T->Boolean)
              (Values: with (T:Type))
  := with (
    T:Type ;
    empty: T ;
    at: T -> T$Keys -> T$Values ;
    update: T -> T$Keys -> T$Values -> T ;
    ... )
```

The parameterized category *DictionaryCat* provides the interface of parameterized packages which receive two arguments: a package with ordering operations on its carrier set (*Keys*) and a package which provides a carrier set (*Values*). The type of the returned package depends on the carriers of the arguments as it can be seen by the signatures of the functions *at* and *update*. In this way, parameterized categories form `dependent products` and their instances are mappings from packages to packages: we call them `functors`.

Ideally, a parameterized category is the type of a parameterized package. However, parameterized categories are used extensively in systems like Axiom to code the categories of Modules or Polynomials over a Ring, etc. [Dave90] [Dave91] which are actually simple Σ-types. Given the category of Polynomials as defined previously we can construct a parameterized package which returns implementations of polynomial types:

```
PolynomialRing(R:Ring,E:OrderedAbelianMonoid): PolynomialCat
  := add (
    R := R;
    e := E;
    Rep := List {k:Rep$E, c:Rep$R};
    leadingCoefficient := r +> ... ;
    degree := e +> ... ;
    ...
  )
```

The advantage of this modeling is that the implementation of the coefficients and the terms of polynomials are accessible from the package which defined functionality on them. In this way, each polynomial package propagates all the information about its constituents: extensions of their functionality is possible by accessing not only the structure, but also the implementation of the types of their components.

### Dependent Products

The category *DictionaryCat* corresponds to the `product type`

$$\Pi(K : Ord).\Pi(T : Set).\Sigma(t : Type).[t, t \to K.t \to T.t, t \to K.t \to T.t \to t, ...]$$

The types of functors are `dependent products` (Burstall and Lampson [Lamp88], MacQueen [Macq84] [Macq86], Harper and Lillibridge [Harp94] Leroy [Lero94]): the type of the return package (domain) can depend on the functor arguments. In the declarations

$$F(m : C') : C = M$$

$$M'' = F(M')$$

$$[\Pi F{-}\text{apply}] \quad \frac{E \vdash m_1 : \Pi\ m : c_1.c_2 \quad E \vdash m_2 : c_1}{E \vdash m_1(m_2) : [m_2/m]c_2}$$

$$[\Pi{-}\text{instance}] \quad \frac{E; m : c \vdash m' : c'}{E \vdash (m : c) \mapsto m' : (m : c) \mapsto c'}$$

$$[\Pi{-}\text{apply}] \quad \frac{E :\vdash m_1 : c_1 \quad E \vdash add(M = m_1, m) : \Sigma M = m_1.c_2}{E \vdash m : (\Pi M : c_1.c_2)m_1}$$

$$[\Pi{-}\text{sub}] \quad \frac{E \vdash c_1' \sqsubseteq c_1 \quad E; m : c_1' \vdash C_2 \sqsubseteq c_2'}{E \vdash \Pi(m : c_1).c_2 \sqsubseteq \Pi(m : c_1').c_2'}$$

$$[\Pi{-}\text{subapply}] \quad \frac{E \vdash \Sigma m' = m_1.c \sqsubseteq \Sigma m' = m_1.[m'/m]c_2}{E \vdash c \sqsubseteq (\Pi(m : c_1).c_2)m_1}$$

Figure 9.3: Subtyping rules for Π-types

where $M$ is the instance of the category $C$, $M'$ an instance of the category $C'$ and the type of the components of $M''$ are equal to:

$$M_i'' : [M'/m]C_i$$

As it has been mentioned, the arguments of a functor (and of its dependent type) are always packages; a functor application results in a new package. The rules for typing and subtyping for dependent products are given in figure(9.3).

In order to pass a value $f$ as argument to a functor, a new anonymous package with a value component $f$ is generated implicitly, resulting in a functor with a package argument:

$$\Pi(f : t_1 \to t_2).M \Rightarrow \Pi(s : \Sigma().t_1 \to t_2).[f\$s/f]M$$

For example, consider the transformation:

$$F(x : Integer) : C := M \Rightarrow F(X : with(x : Integer)) : C := [x\$X/x]M$$

If we call $F$ with an integer argument $F(x := 3)$, then an anonymous package is generated: $F(add(x : Integer := 3))$ and $M$ is returned. If we pass two arguments, as in $F(x := 3, y := 4)$, then following the same procedure we have

$$F(add(x : Integer := 3, y : Integer := 4))$$

Given the rules ΠF-apply and Σ-forget, the new argument matches the type $\Sigma().[x : Integer]$, and the functor application returns $M$.

### Subtyping of Products

As we have seen, the arguments of parameterized categories have sum types, while the body is also an instance of a sum type. The case of categories (and domains) with $N$ arguments is viewed recursively: the body is parameterized by $N - 1$ arguments, i.e. it has a product

type, and after $N$ steps the final body is not parameterized (0 arguments), resulting in a sum type.

The subtype relation among sum types can be extended to accommodate dependent products. The subtyping relation in this case involves *contravariance* in the argument position and *covariance* in the body type. The type $\Pi(m : C_1).C_2$ places a stronger constraint on the behaviour of its instances than $\Pi(m : C'_1).C'_2$ if it demands that:

1. they behave properly on a larger set of inputs ($C'_1) \sqsubseteq C_1$) or
2. their results fall within a smaller set of outputs ($C'_2 \sqsubseteq C'_2$), provided that the inputs range in $C'_1$

or possibly both ([Π-sub]).

### Sharing (Take Two)

Sharing among type components is of special importance in the case of functors and product types. The rule Σ-transparent for sharing is powerful enough to allow the definition of functors which combine or compose domains and packages:

```
RingFun (P1:Monoid) (P2: with (Monoid; Rep=P1.Rep)):Category :=
  with ( Monoid ;
         * : Rep - Rep -> Rep ;
         1 : Rep )

BuildRing: RingFun := P1 +> P2 +>
  add ( P1;
        * := (+)$P2 ;
        1 := 0$P2 )
```

In this example, two monoids with the same representation are combined in order to form a ring. Since the category *Monoid* already exists in the system, the only additional action is to add a sharing constraint to it for $P2$.

Without sharing the above construction is more complicated and less natural for modeling algebraic concepts. It would require the parameterization of algebraic categories with their representation:

```
RingFun(T:Type)(M1:Monoid(T))(M2:Monoid(T)):Category := ...
```

Such constructs demand the implicit or explicit creation of new hierarchies for parameterized monoid categories. The former increases the load of a type system and its implementation dramatically while the latter demands that the user create parallel hierarchies for all possible cases of parameterization.

Sharings reduce the need for instantiation of parameterized categories, as the above example shows. For instance, many cases which involve parameterized categories in the Axiom library can be reduced to simple categories with sharing constraints. The rest can be transformed to sum types with sharing constraints on their type components. This feature can be exploited for the implementation of higher order categories by means of first order possibly nested constructs and sharings, which reduce the complexity and the computational load of the compiler.

### Flattening Dependent Products

The application of dependent products for the generation of new sum types complicates the elaboration of programs and adds considerable load in the typing process. In the presence of subtyping, new hierarchies have to be created since product types may extend other products. Sums do not have such complications and their hierarchies remain stable.

One of the properties of Π-types is that their components cannot be accessed, thus sharing constraints cannot propagate to their environment. Therefor it is always desirable to involve Σ-types in the presence of static typing. On the other hand, sharing specifications can restrict a sum to certain implementations, emulating the results of product application to equivalent implementations.

Another observation we made in the last sections is that it is not always convenient to implement categories as products if we want to access the implementations of the constituents of a package. For instance, the code of DictionaryCat was rather artificial. A more realistic implementation would demand the type of the keys and values to be accessible so as to perform computations other than the ones specified in this category, like transforming dictionaries to other structures. The category

```
DictionaryCat(Keys:=String; Values:=Integer)
```

could be defined as

```
DictionaryCat := with (
  Keys: with(Type;...);
  Values: Type;
  Rep: Type;
  at: T -> Rep$Keys -> Rep$Values
  ...
  );
  with (DictionaryCat; Keys=String; Values=Integer)
```

allowing dictionaries to expose the implementation of Keys and Values and propagate evntual sharing constraints.

However, in some cases, instantiation of dependent products can serve modeling purposes better than dependent sums. For instance, a *Ring* is a *LeftModule* over itself:

```
LeftModules(R: with(AbelianGroup; SemiGroup)) :=
  (AbelianGroup;
   * : (Rep,R) -> Rep )
Ring := with (LeftModules(%):...)
```

where the construct % refers to the package which is an instance of the category we define (in this case *Ring*). If $R$ were a nested package inside the definition of LeftModules then it should be defined forall the Rings too. This would cause great inconveniences in the coding of Rings since a Ring is a package which has an $R$ component assigned to itself, leading to recursions. The typing system should handle these cases automatically and in a transparent way.

The application of dependent product types can be defined in terms of already defined constructs such as nested packages and sharings as the previous examples indicate. We distinguish two cases of interest:

1. Declarations of packages as instances of applications of Π-types of the form

$$P_2 : (\Pi(P : C_1).C_2)P_1 := P_3$$

2. Inheritance of applications of Π-types of the form

$$C_3 := \Sigma((\Pi P : C_1.C_2)P_1).C_x$$

the right hand side of which is essentially the expression

$$with((\Pi(P : C_1).C_2)P_1; C_x)$$

The expression $(\Pi(P : C_1).C_2)P_1$ creates a sum type $\Sigma P : C_1 = P_2.C_2$ with the sharing specification $P = P_2$.

This transformation has another important consequence: it allows products to be transformed into sums, meaning that the arguments of a Π-type can have Π-types themselves without changing their inital semantics. This makes possible the definitions of `higher order functors` similar to Cregut and MacQueen [Macq94].

The RHS of the expression $P_2 : (\Pi(P : C_1).C_2)P_1 := P_3$ is handles as above, but the typing mechanism should automatically add a $P$ component to $P_3$ so that $P_3$ will match with the resulting sum type. Consequently, $P_2$ will have an implicit $P$ component. In case the declaration specifies the argument of a dependent product then $P_3$ plays the role of the argument passed during the application of the product as in $(\Pi(P : (\Pi(P : C_1).C_2)P_1).C_3)P_3$

The expression

$$\Sigma((\Pi P : C_1.C_2)P_1).C_x$$

is transformed to

$$((\Pi P' : C_1.\Sigma[P'/P]C_2.C_x)P_1) \quad \text{where } P' \text{ fresh}$$

and then we proceed as above.

These transformations assume that such directives to the typing mechanism should be coded directly in the Π-types and then elaborated during their applications, so that the appropriate components of the resulting Σ-types and their instances will be introduced when needed. This involves the following steps:

1. Add a directive for introducing component $p$ in the definition of the instances of the category resulting from applying $\Pi(P : C_1).C_2$ to a package $P_1$. In order to avoid name conflicts, a fresh variable $P'$ can be introduced instead of $P$, performing the appropriate $\alpha$-conversions.
2. Create a dummy package $P_{dummy}$ which satisfies $C_1$
3. In the definition of the sum $\Sigma P' : C_1.[P'/P]C_2$ propagate the directive of step 1 and do type-checking by sharing $P_{dummy}$ from step 2 with $P'$. The directive from step 1 should now affect the instances of the sum.
4. If type checking succeeds, remove $P_{dummy}$. When a domain is matched against the introduced sum, the stored directives introduce the appropriate nested packages automatically, resulting in the generation of a new package. Add the appropriate sharings by generating a new category.
   In the above example,, if a package $M$ is to be matched against $Module(\%)$ (where % refers to $M$), a new sum type $Module_{flat}$ is generated through steps 1-3. Consequently a package $M_{new}$ is generated including all the components of $M$ plus a component $P'$ resulting from the directives of step 3. A sharing specification $R' = M$ is added to $Module_{flat}$. Type checking and propagation of sharings proceed.

5. Any other operation involving sums should take into account the directives stored in them, or inherited by other sums. In this case, *Ring* should include the directives given by *LeftModule(%)*.

The above result in the rules Π-apply and Π-subapply in figure(9.3).

The typing process in this scheme is decidable, due to the generation of new sums and packages from existing components. However, the above steps avoid the generation of new type components since the old components are also members of the new packages. Thus propagation of type information is not affected.

The rules for application of Π types imply that a dependent product can be applied only when the dependencies of the body of the product to its arguments have been removed. This assumes an implicit coercion of the `dependent product type` to an `arrow type` as in

$$(m : (\Pi M : C_1.C_2))m_1 \Rightarrow (m : C_1 \rightarrow [m_1/M]C_2)m_1$$

provided that $m_1$ can be shown to have the type $C_1$.

### 9.0.9 Subtyping of Domains

Descriptions of `sets` of entities which belong to various domains can be arranged into useful classification hierarchies. For example, the set of integers can be seen as a subset of the rational numbers. This supports a useful kind of reasoning: if $X$ is an integer, then $X$ must also be a rational number, and every interpretation of a rational number should be true for $X$ if both domains are viewed as instances of the same category. For example, it should be valid to make the judgement

$$\frac{E \vdash \mathit{Integer} \sqsubseteq \mathit{Rational} : \mathit{Ring} \quad E \vdash x : \mathit{Integer}}{E \vdash x : \mathit{Rational}}$$

The semantics of $S \sqsubseteq T$ for domains and types can be derived by the corresponding relations in Σ and Π-types; they are included in the following statement:

If $S \sqsubseteq T : C$, then an instance of $S$ may safely be used in any operation specified in $C$ where an instance of $T$ is expected.

For example the function

```
foo(x:Rational) := (x+1) * x**2
```

can be safely applied to the integer argument 4 because it is possible to view any integer as a rational in the above operations. More important, since integers form a `subring` of rationals, and the definition of *foo* includes operations defined for rings, we can do this substitution under the assumption that both *Integer* and *Rational* form a ring. This leads to the more general subrule [sub-val] (figure 9.4).

#### Coercions

Two formal accounts can be given for the semantics of subtyping. In the simpler view, the syntactic subtype relation $S \sqsubseteq T$ where $S, T$ are domains, is interpreted as asserting that the `semantic domain` denoted by $S$ is included in that denoted by $T$. In the more general view $S \sqsubseteq T$ is interpreted as a `canonical coercion function` from the domain denoted

by $S$ to the one denoted by $T$:

$$\frac{E \vdash S, T : C \quad E \vdash x : S \quad E \vdash \mathbf{coerce} : S \hookrightarrow T}{E \vdash \mathbf{coerce}(x) : T}$$

i.e.

$$\frac{E \vdash x : S \quad E \vdash {}_C\mathbf{coerce} : S \hookrightarrow T}{E \vdash {}_C\mathbf{coerce}(x) : T}$$

where coercions keep information about the form of subtyping among domains in terms of algebraic relations: if $C$ involves *Ring* structures, then the above rule describes a subring relation.

Since coercions can be composed, the subtype information carried with them can be modified. In a type system, information cannot be generated from inside, therefore any composition of coercions reduces the information:

$$\frac{E \vdash C_1 \sqsubseteq C_2 \quad E \vdash {}_{C_1}\mathbf{coerce} : R \hookrightarrow S \quad E \vdash {}_{C_2}\mathbf{coerce} : S \hookrightarrow T}{E \vdash {}_{C_1}\mathbf{coerce} : R \hookrightarrow T}$$

$$\frac{E \vdash C_1 \sqsubseteq C_2 \quad E \vdash {}_{C_2}\mathbf{coerce} : R \hookrightarrow S \quad E \vdash {}_{C_1}\mathbf{coerce} : S \hookrightarrow T}{E \vdash {}_{C_1}\mathbf{coerce} : R \hookrightarrow T}$$

In this way paths can be composed with different subtyping carried information. A useful theorem which can be proved by the above rules is that any direct coercion between two domains, carries more information than any other coercion path which uses an intermediate domain. This means that direct coercions are preferable in case of multiple paths. Assume the domains $A$, $B$, $C$, and $D$, and the coercions

$$A \hookrightarrow C \hookrightarrow B$$
$$A \hookrightarrow D \hookrightarrow B$$

A type system cannot in general prove whether this graph commutes, since the former path may have different semantics than the latter. Since the path $A \hookrightarrow B$ keeps more information than the path $A \hookrightarrow C \hookrightarrow B$ any composition of the above paths leads to similar results, i.e. the path

$$A \hookrightarrow B \hookrightarrow D$$

is preferable for a transition from $A$ to $D$ than

$$A \hookrightarrow C \hookrightarrow B \hookrightarrow E \hookrightarrow D$$

It is important to mention that the length of the path is of no particular importance, although the preferable path is always shorter than any other path. For instance, if in the above paths the coercion $C \hookrightarrow D$ is added, then there is no way to choose any particular path.

### Subtyping of Representations

In this section we restrict ourselves to natural subtyping between types of run-time objects. The subtyping rules for records, arrow types, and recursive types correspond to the subtyping rules for packages ($\Sigma$-thinning), dependent products ($\Pi$-sub), and domains ($\exists$-forget); a similar rule for `disjoint unions` is added (figure 9.4).

$$[\text{sub}-\text{val}] \quad \frac{E \vdash x : S \quad E \vdash S \sqsubseteq T : C}{E \vdash x : T}$$

$$[\{\}-\text{sub}] \quad \frac{\forall j \in \{1,\ldots,m\} \exists i \in \{1,\ldots,n\} \;.\; E \vdash x_i \equiv x_j \;\; E \vdash \tau_i \sqsubseteq \tau_j, \;\; m \leq n}{\{x_i : \tau_i\}_{\forall i \in \{1,\ldots,n\}} \sqsubseteq \{x_j, \tau_j\}_{\forall j \in \{1,\ldots,m\}}}$$

$$[\rightarrow -\text{sub}] \quad \frac{E \vdash T_1 \sqsubseteq S_1 \quad E \vdash S_2 \sqsubseteq T_2}{E \vdash S_1 \rightarrow S_2 \sqsubseteq T_1 \rightarrow T_2}$$

$$[\mu-\text{sub}] \quad \frac{E; T_1 : Type; S_1 \sqsubseteq T_1 \vdash S \sqsubseteq T}{E \vdash \mu S_1, S \sqsubseteq \mu T_1, T}$$

$$[+-\text{sub}] \quad \frac{\forall i \in \{1,\ldots,m\} \exists j \in \{1,\ldots,n\} \;.\; E \vdash x_i \equiv x_j \;\; E \vdash \tau_i \sqsubseteq \tau_j, \;\; m \leq n}{+[x_i : \tau_i]_{\forall i \in \{1,\ldots,m\}} \sqsubseteq +[x_j : \tau_j]_{\forall j \in \{1,\ldots,n\}}}$$

Figure 9.4: Subtyping rules for types

Records can be viewed as degenerated forms of packages where there are no type components, and no dependencies among the components and their types. The above rule for records supports the subtyping of tuples if we view tuples as records with integer fields.

In a similar way, an arrow type is a Π-type where there is no dependency between the input type and the output: $\Pi(x : \tau_1).\tau_2$ reduces to $\tau_1 \rightarrow \tau_2$ if $x$ is not in the expression $\tau_2$.

The rule for function types interprets the rule for records in the following way: If we think of a record as a function from labels to values, a record $\tau$ represents a stronger constraints than $\tau'$ on the behaviour of such function, if $\tau$ describes the function's behaviour on a larger set of labels or gives a stronger description of its behaviour on some of the labels also mentioned by $\tau'$ (Pierce [Pier91a]).

By means of [+-sub] and [$\mu$-sub] our type system can assure that

$$Tree(Integer) \sqsubseteq Tree(Rational)$$

.

For continuing our analysis we introduce the category *Ring* and repeat the definition of the category *Field*:

```
Ring := (Rep:Type) +> with (
  + : Rep -> Rep -> Rep ;
  - : Rep -> Rep ;
  * : Rep -> Rep -> Rep ;
  0 : Rep ;
  1 : Rep )

Field := (Rep: Type) +> Ring(Rep) with (
  / : Rep -> Rep -> Rep )
```

and $Field \sqsubseteq Ring$ given the subtyping rules for existential types above. This means that *every instance of the category Field can be viewed as an instance of Ring* or *every instance of an instance of Field is also an instance of an instance of Ring*. Unfortunately the rules for subtyping among categories cannot be easily extended to rules for their instances.

Given the definition of the type *RationalFun* from above we can introduce in a similar way the type of integers which form a Ring. Our notation allows us to abstract the implementations away:

```
IntegerFun := (Rep:Type) +> (
  new: Integer -> Rep ;
  numerator: Rep -> Integer ;
  denominator: Rep -> Integer ;
  SetNumDenom: Rep -> Integer -> Rep ;
  + : Rep -> Rep -> Rep ;
  - : Rep -> Rep ;
  * : Rep -> Rep -> Rep ;
  0 : Rep ;
  1 : Rep ; )
```

It is not the case that $IntegerFun \sqsubseteq RationalFun$ since the specification of integers does not include the record field /, leaving $IntegerFun$ with one field less than $RationalFun$.

The obvious solution to cope with the above problem is the use of coercions, as we saw in the previous section. The problem is that coercions are expensive in computational resources and in some cases they can introduce inconsistencies.

### 9.0.10 Type Classes

In this section, we are concerned with modeling subtyping without use of coercions. We introduce the concept of type classes, which has some similarities with the homonymous concept in Haskell (Nipkow and Prehofer [Nipk95], Wadler and Blott [Wadl88]). The semantics and the formal definitions for type classes are provided in the next section. Type classes are distinct from categories in the sense that they are not part of the user defined types. The main difference with Haskell's type classes is that our type classes are not syntactically defined in the language but are inferred by its type system, reducing the complexity of the former.

It is clear now that we are not interested in a pure subtype relationship, but in an algebraic relationship based in terms of `subrings`, `subfields`, etc. We introduce type classes in order to provide the facility of viewing Integer as a `subring` of Rational:

```
RationalRing := (Rep:Type) +> with (
  + : Rep -> Rep -> Rep ;
  - : Rep -> Rep ;
  * : Rep -> Rep -> Rep ;
  0 : Rep ;
  1 : Rep )
```

This declares that any eventual subtype of rational which happens to be an instance of *Ring* implements the operations declared in the type class *RationalRing*. Since *Integer* forms a subring of *Rational*, from the definition

```
IntegerRing := (Rep:Type) +> RationalRing(Rep)
```

we may conclude:

$$IntegerRing \sqsubseteq RationalRing$$

The declaration of $Integer$ as instance of the type class $IntegerRing$ is straightforward:

```
Integer :: IntegerRing
```

while for Rationals we need one additional type class:

```
RationalField := (Rep:Type) +> RationalRing(Rep) with (
  / : Rep -> Rep -> Rep )

Rational :: RationalField
```

Given the function definition

```
dblSqrd(x::Rational) := (x+x)*(x+x)
```

by means of type classes the system infers the most general signature for *dblSqrd*:

$$\Gamma, \forall a :: RationalRing \vdash dblSqrd : a \to a \to a$$

*dblSqrd* can receive as argument an instance of any the types of *RationalRing*, including *Integer*, without any need to coerce it to *Rational*: we have managed to define a natural subtype relationship between integers and rationals, which comes in accordance with the algebraic semantics of the terms.

### Definition of Type Classes

For the definition of type classes we assume that operations do not belong to a type but to an algebra (that is, a particular collection of types). A class combines one or more types for the implementation of its instances. A class's instances do not need to have a common internal structure but they are elements of the types which a class assumes. Herewith we can define type classes.

Formally a `type class` has the following structure: $Class[T, B]$ in which $T$ is the set of types and $B$ is the behaviour of the class's instances. An instance of a type is by definition an instance of any of the classes in which this type belongs. The `instanceOf relation` (denoted by ::) represents membership in a set of instances and as such it is *irreflexive* and *non-transitive*.

The `subclass relation` (denoted by $\sqsubseteq$) is a *reflexive*, *antisymmetric*, and *transitive* binary ordering relation in the partial ordered set of classes.

`Subtyping` can be seen in two ways (which are consistent with the definitions given in the previous sections): subtyping by means of subclassing or coercions.

$$[\bigcap\text{-sub}] \quad \frac{\forall j \in \{1,\ldots,m\} \exists i \in \{1,\ldots,n\} \;.\; E \vdash \tau_i \sqsubseteq \tau_j,\; m \leq n}{\bigcap(\tau_i)_{\forall i \in \{1,\ldots,n\}} \sqsubseteq \bigcap(\tau_j)_{\forall j \in \{1,\ldots,m\}}}$$

Figure 9.5: Subtyping rules for intersection types

$$\frac{E \vdash x :: C \quad E \vdash C \sqsubseteq C_i}{E \vdash x :: C_i} \qquad \frac{E \vdash x :: C \quad E \vdash \mathbf{coerce} : C \hookrightarrow C_i}{E \vdash \mathbf{coerce}(x) :: C_i}$$

The above rules handle the case of multiple subclassing for they are applied $\forall i \in [1,\ldots,n]$.

Type classes $C_1$, $C_2$ are said to belong to the same `inheritance path` when one can derive through $\sqsubseteq$ or $\hookrightarrow$ relationships that $C_1 \sqsubseteq C_2$ or $C_1 \hookrightarrow C_2$ respectively.

The above inductive definitions can be seen as the definition of `class intersection` which differs from the classical definition of set intersection in the sense that

equality for instances of different classes cannot be established. Such a relation is possible only between members of the same equality type. Identity is only possible between members of the same type. Class intersection corresponds to intersection of the sets of types implementing the classes. Classes define behaviour by means of a set of axioms and operations among each class's instances, therefore a class intersection produces behavioural *union*. This definition has constructive power since an instance must be an element of a particular type. It corresponds to the subtyping rules for `records` (union), and `intersection types` (figure 9.5).

`Class Intersection:`

$$x :: (C_1 \sqcap \ldots \sqcap C_n) \Leftrightarrow \forall_i x :: C_i$$

and

$$[T_1, B_1] \sqcap \ldots \sqcap [T_n, B_n] = [\bigcap_i T_i \bigcup_i B_i]$$

where we write $[T, B]$ for the class implemented by each of the types $t \in T$ and supporting behaviours $b \in B$.

In the case that $T_1 \cap \ldots \cap T_n = \{\}$, the class intersection is $\bot$. Similarly, we can define the union of classes as their superclass:

`Class Union:`

$$x :: (C_1 \sqcup \ldots \sqcup C_n n) \Leftrightarrow \exists_i x :: C_i$$

$$[T_1, B_1] \sqcup \ldots \sqcup [T_n, B_n] = [\bigcup_i T_i, \bigcap_i B_i]$$

The rules we use for typing values with type-classes are similar to those of the Haskell system. We avoid repeating them here since they can be found in (Nipkow [Nipk95]). Every new value declaration which involves instances of a domain which is a member of a type class is added to the value set of this type-class. Since a domain can be a member of many type classes the most general one which ensures the typing of the value is chosen. For every new value the same process is followed. This means that although the initial constructions of type classes depends on the category hierarchy, after the introduction of new values they can include more declarations.

### Coercions vs. Typeclasses

Typeclasses allow a different kind of behaviour than coercions. If there is a coercion from $A$ to $B$, but we can perform the operations defined in $B$ directly in $A$ (assuming that instances of type $A$ are passed as arguments) then we do not need to consider the possibility of multiple paths or computational effects, but we get the behaviour defined in $A$ (instead of that in $B$). The former is an advantage of typeclasses over coercions, the latter allows for more options in the design.

It is worth noting that none of the forgoing would have been possible if the type variable *Rep* was not introduced in the definition of domains and categories. Suppose that Rational and Integer had the definitions

```
Rational := ( ...
  + : Rational -> Rational -> Rational
  etc. )

Integer := ( ...
```

```
+ : Integer -> Integer -> Integer
etc. )
```

This would influence the definitions of RationalFun and IntegerFun respectively, and finally the definitions of RationalRing and IntegerRing. There would have been no simple way to derive a subtype relationship among types which would include the terms

$$Rational \rightarrow Rational \rightarrow Rational$$

and

$$Integer \rightarrow Integer \rightarrow Integer$$

since these two cannot form a subtype relationship due to the introduction of the `contravariance` rule for function types.

For the same reason it is also difficult to use packages for defining the above form of subtyping. Domains have only one carrier set, making it simple to reason and infer type hierarchies. In the presence of more than one combined sets this task is more difficult since it can involve recursive domain definitions. Additionally, the form of subtyping proposed can be easily implemented with dictionaries specific to each type class. In the presence of multiple carriers this implementation requires multiple dictionaries, most of which may never be used. This approach would decrease the efficiency of the system without acutally bringing any substantial benefit.

### 9.0.11 Comparison with Related Work

The type system we have presented is an extension of Axiom's type system with the introduction of strong sums, higher order functors, transparent category types, structural type equivalence, sharing specifications and type classes. It can also be viewed as an extension of SML's module system by transforming abstract types (domains) to modules (packages), and introducing recursive higher order dependent products.

Work of considerable value on type systems for computer algebra was done during the 90's with examples such as *Newspeak* (Foderado [Fode83]). A comparative review of systems of that generation is given by Fateman [Fate90]. An implementation which increased our understanding about the relation between abstract types and existential types was the *SOL* language by Mitchell and Plotkin [Mitc88] and an extension of *SML* with subtyping (Mitchell, Meldal, and Madhav []). Also the theorem prover *Nuprl* based on predicative logic can elaborate many parts of our type system. $\Sigma$-types are part of the *Nuprl* system and facts about abstract theoretical levels can be proved directly.

Axiom's new language (Watt [Watt94a]) includes the two first extensions but not the last three. Its full type system unifies the concept of domains and packages into the concept `type`, allowing objects to be instances of packages. This introduces syntactic sugar for expressions of the form $Rep\$P$ we described above and is easily handled by a compiler. It views packages as records, something which needs delicate handling for avoiding unsoundness. Thus categories ($\exists$ or $\Sigma$ types) are considered as types too, disallowing the use of the [forget] rule, since this would lead to undecidability. In fact, sharings are not supported.

Various systems handle polymorphism in different ways. Axiom allows parameterization of domain definitions. In addition, its new language uses types as first class values (as in the language *Pebble* by Burstall and Lampson [Lamp88]) and adds run-time type tests, meaning that types can be passed as function parameters or returned by functions dynamically. This

involves loss of static typing information as in (Mitchell, Meldal, and Madhav [Mitc88]) where $\Sigma$-types are coerced to $\exists$-types. In a language which does not allow overloaded identifiers for values in a given context, type variable can be inferred automatically as in the Hindley-Milner type system or by means of type classes as in *Haskell's* typing mechanism, allowing for implicit type parameterization of functions. In *XFun* by Dalmas [Dalm92] everything can be a first class value as in the new Axiom, even categories (called `signatures`). *XFun* relies on run-time type checks. The same is valid about the *Magma* system by Cannon and Playoust [Cann01], which, however, does not include many of the features we presented. *Magma* does not allow user defined types.

*SML* supports functors with `transparent signatures` and *SML/NJ* extends their definition to higher order functors without mixing them with values. Recent extensions of the *SML* module system with `translucent sums` by Harper and Lillibridge [Harp94] involve the definition of `first class higher order modules` allowed to be defined as values. Type-checking restricts their flexibility in type contexts in order to maintain static and strong typing. Our system does not support the use of types as values but allows functors and products to be applied to them. The concept of type classes has been inspired by *Haskell* in order to allow overloading of user defined functions on arbitrary types. The extension to support modules as values is possible since modules consist of a compile-type and run-time part as described in Harper, Mitchell, and Moggi [Harp89]. All type information is resolved at compile time but implementations can be computed at run-time, thus type components cannot be used as values. For instance, if the body of a function $f$ with argument $x$ involves code which applies a functor $F$ to a package $D$ the value components of which may depend on $x$, then the part of $F$ which involves type components (static part) can be applied to the type components of $D$ out of the body of the function $f$, and the value part of $F$ can be applied to the value components of $D$ inside the body of $f$, assuming type information derived from the static part. If the types depend on $x$ then only abstract types can be created, i.e. only minimal typing information can be propagated. Use of categories (or types) as values in a static type system is reasonable only under restrictions which allow for dynamic typing.

An advanced concept which is part of the *Nuprl* inferencer is the use of `propositions as types`. Using propositions, properties like associativity can be expressed and proved at the category level but operations like equality demand special treatment. We have not addressed this issue in our presentation because we want first to establish basic type theoretic properties and examine the limits of the extensions analyzed. Additionally, the application of propositions is not so clear for modeling algebraic structures. Expressing category properties in a more limited form as in Axiom can be easily incorporated in our system by introducing additional (type or value) components into categories.

The sharing construct we presented is an extension of *SML*'s sharing constraints (MacQueen [Macq88], Milner and Tofte [Miln91]) and their extensions by Leroy [Lero94], and Harper and Lillibridge [Harp94], since we allow arbitrary type constructors to be shared, to the point this is allowed by the syntactic rules we defined in the introduction. Domains are implemented by the construct *abstype* in the core *SML* language, or by using packages (abstractions) with opaque types in the *SML* module system. This is not possible for functors which always have transparent types.

A difference between *SML* and the system presented in this paper is that functors in *SML* are `generative`, i.e. two instantiations of a functor do not necessarily result in the same type or package. This implies that two occurrences of *Polynomial(Integer,x)* in the same context may not have compatible carrier sets.

### Conditional Categories

A feature of Axiom which we have not elaborated in this paper are `conditional categories`, that is, parameterized categories whose body size and kind depends on their arguments. The introduction of dependent sums would allow conditional dependencies on the body of the category which carries them.

Although the type theoretic properties of conditional categories are not well understood, it is easy to show that without certain restrictions imposed on them they result in undecidability quite easily, even if we assume that domains are not first class objects and the arguments of a category are not first class values (the undecidability in the case of value arguments is obvious and has been examined by Weber [Webe93b]). This happens because the type of the argument of a conditional category (and thus its body) can increase in the presence of recursive definitions in a way similar to that encountered at the $F_{\leq}$ type system which has been shown by Pierce [Pier91a] to be undecidable. For example, given the definitions:

```
C():Category == with (if % has Ring then Ring)
t: C() == add ...
```

it is not clear whether $t$ is a *Ring*, and the system may loop. The above situation is handled in *Gauss* (Monagan [Mona93]) purely syntactially by viewing % as the category in scope and checking if the defined category (here C) includes code for Rings up to the point where the conditional is found: this would add *Ring* to $C$.

Despite such inconveniences, conditional categories are a very useful and powerful construct, which is able to simulate examples supported by other higher order systems (like $F_{\leq}$), and can implement in a unique way many applications from computer algebra where it can increase our understanding about the interaction of types.

## 9.0.12 Conclusions

We have presented selected parts of a type system for symbolic computation which is sound and consistent, and powerful enough for algebraic applications since it combines and extends commonly used systems. The issues of abstract representations, abstract and concrete implementations, subtyping without coercions and sharing of domains have been handled, solving many problems of existing approaches. For the purposes of our analysis we have extended the concept of categories to include strong sum types and sharings, and used existential types for defining subtyping among domains without need of coercions.

A type inference mechanism transforms domains into packages and flattens parameterized categories with package instances into sum types with sharings. In this way we avoid the combinatorial explosion of subtyping hierarchies among categories. The information about domain structures is saved for later stages of the inference process. Another mechanism constructs type classes as a combination of categories and domains, in order to resolve the conflicts introduced by the different definitions of subclass and subtype in algebra and type theory accordingly. However, in cases in which a subdomain does not define operations of its supertype, coercions can be called.

We tried to concentrate on typing issues with respect to computer algebra rather than presenting a complete language or system design. Therefore we left aside other important topics such as exception handling, pattern matching, etc. although these constructs are subject to typing in our scheme.

Additional research is of interest in the following subjects:

1. Introduction of `first class packages` in the system without sacrificing the nice properties of the presented stratification. This should split packages and functors into a static and dynamic part. In such case, records and functions can be removed as special cases of higher order constructs. The difficulties here involve recursive definitions and effects. We need a clear semantics for recursive package definitions.
2. Examination of the appropriate constraints for the introduction of conditional categories in order to maintain soundness.
3. An efficient implementation of type classes without sacrificing space
4. A scheme for coercions which involes algebraic relations as in the case of the *Weyl* system by Zippel [Zipp93] instead of simple coercions from one type to another.
5. A more remote target is the introduction of propositions at the category level and the integration of an environment for theorem proving at the top level.

This introduced type system is currently implemented for the $\lambda A$ language designed and implemented by the author at ETH Zurich. The concepts presented have been tested in our implementation by translating them to SML constructs.

# Chapter 10

# Doye's Coercion Algorithm by Nicolas Doye

This chapter is based on a PhD thesis by Nicolas James Doye [Doye97]. Changes have been made to integrate it.

## 10.1 Introduction

### 10.1.1 Abstract Datatypes in General

Abstract datatypes provide a way of specifying a type. As a (useful) side effect an abstract datatype can express how a whole collection of types may act.

An incredibly specific specification will only specify types which are all "the same" (isomorphic). A less exacting specification may result in types which are similar, but not the same.

This "side-effect" of using a less exacting specification to collect similar types together forms the basis of strict categorical type systems.

For example, we may say which operations are available on a certain family of types. We may also state certain facts about the structure of all types in a certain family. Most importantly of all, we can enforce relationships between various types (of different families), how they interact, and how they may depend upon each other (or not).

See section 10.4.1 for how we may specify types and what operations are available to them. Look at section 10.4.48 for the relationships between different types. Section 10.4.28 details how we may restrict the structure of certain types using sets of equations.

As a simple example, a polynomial ring depends for its specification on the underlying ring (from which the coefficients are taken).

As a more complicated example, a polynomial ring could also depend upon: the type from which it takes its variables; the type of the exponents; a boolean algebra; an ordered free monoid with one generator (the natural numbers adjoin {0} is often a good choice); and maybe a few others.

Another factor is the actual relationship between the various types. In our above example, of the polynomial ring, we know that:

1. there is a ring monomorphism from the underlying ring to the polynomial ring
2. there is an injection from the variable type[1] to the polynomial ring. (In fact, an ordered-set monomorphism[2].)

Other types often have more complicated lattices of relationships. These relationships are often able to be abstracted out to apply all the instance types of a particular abstract data type.

### 10.1.2 The Problem

A commonly encountered difficulty in languages which utilise types is the following:

1. Given `thing`, of type `Type1`, can we change the type of `thing` to be that of `Type2`? More accurately, can we create an element of `Type2`, which corresponds, in some natural way to `thing` of type `Type1`?
2. If so, how do we go about performing such an operation? Can we perform this task algorithmically?
3. Does there exist a way of abstracting this question or must every (`Type1`, `Type2`) pair be considered? Moreover, can we abstract such an algorithm (as mooted in 2, above) out to cover all cases?

Such type changes are often called *coercions*, *conversions*, or *castings*, with all three words having slightly different meanings.

In this work, we will consider castings and conversions to mean any type change, regardless of mathematical rigour. We will call a type change a coercion if it is in some way, "natural". We will define this more rigourously later.

Computer algebraists often need to use coercions since informally most mathematicans alter the domain of computation without saying phrases like, "in the monomorphic image of $A$ to $B$" or, "forgetting that $x$ is a $T$ and viewing it as a $U$".

For example, we often view the integers ($\mathbb{Z}$) as polynomials (in $\mathbb{Z}[x]$) rather than considering the constant polynomials of $\mathbb{Z}[x]$ as being a monomorphic copy of $\mathbb{Z}$. This is the coercion $\mathbb{Z} \to \mathbb{Z}[x]$.

Other simple examples of this sort of coercion are:

$$\mathbb{Z} \to \mathbb{Q}, \quad \mathbb{Z}[x] \to \mathbb{Z}[y,z], \quad \mathbb{Z} \to \mathbb{Q}[x,y], \quad S(2) \to S(5), \quad \mathbb{Z}[x] \to \mathbb{Z}(x)$$

where $S(n)$ is the symmetric group on a set of $n$ symbols.

Coercions are useful when creating elements of quotient structures, such as $\mathbb{Z} \to \mathbb{Z}_n \cong \mathbb{Z}/n\mathbb{Z}$ where $(n \in \mathbb{N})$. Other epimorphic (surjective) examples include,

$$\mathbb{Z}_{25} \to \mathbb{Z}_5, \quad G \to G/G'$$

where $G$ is any group and $G'$ is the commutator subgroup of $G$.

[1] The type from which all the variables are taken. In Axiom, the variables are usually elements such as X, Y, and Z which are from the type Symbol

[2] This is an injective function, $\phi$, which is both a set-homomorphism (which is just a total function) and preserves order. So if $a < b$ then $\phi(a) < \phi(b)$. Mathematicians often utilise the order on the variables and extend it to an order on polynomials

Under certain circumstances, coercions can be "lifted" into other constructors. We refer to these as "structural coercions" later in this work. As examples consider

$$List(\mathbb{Z}) \to List(\mathbb{Q}), \quad List(\mathbb{Z}_{25}) \to List(\mathbb{Z}_5), \quad \mathbb{Z}_{25}[x] \to \mathbb{Z}_5[x], \quad \mathcal{M}_{2,2}(\mathbb{Z}) \to \mathcal{M}_{2,2}(\mathbb{Z}(x))$$

Examples of type-conversions which are *not* coercions include,

$$\mathbb{Q} \to \text{Float}, \quad \text{List}(\mathbb{N}) \to S(n), \quad \mathbb{Z}_5 \to \mathbb{Z}, \quad \mathbb{Z}_5 \to \mathbb{Z}_3, \quad \mathbb{Z}[x] \to \mathbb{Z}[y]$$

Natural type changes (coercions) can be created in the interpreter at the moment using a series of *ad hoc* measures. All of this occurs transparently to the user and works for most common types. However, it is *not* algorithmic. One of the aims is to provide an algorithm (algorithm 10.7.11) to create unique (theorems 10.6.30, 10.7.20) coercions.

### 10.1.3 Examples of how Axiom coerces

Axiom knows (or believes) that all functions called `coerce` are coercions and will use them to "build" other coercions using certain rules, and special cases defined in the interpreter. Here we give examples of how Axiom coerces between certain common types.

#### Simple examples

**Example 10.1.4** $\mathbb{Z} \to \mathbb{Q}$

In Axiom, $\mathbb{Z}$ is of type `Integer` and $\mathbb{Q}$ is of type `Fraction(Integer)`. The type constructor `Fraction(R : Ring)` exports a function[3]

$$\text{coerce} : \text{R} \to \%$$

which is the natural monomorphism including the ring `R` in its fractional field `Fraction(R)`.

**Example 10.1.5** $\mathbb{Z} \to \mathbb{Z}[X]$*:*

In Axiom. the type `UnivariatePolynomial(X,Integer)` typically represents $\mathbb{Z}[X]$. This is constructed by the type constructor,

```
UnivariatePolynomial:(S:Symbol,R:Ring) ->
   PolynomialCategory(R,NonNegativeInteger,Symbol)
```

**UnivariatePolynomial** exports a function

$$\text{coerce} : \text{R} \to \%$$

which is the natural monomorphism including the ring `R` in the polynomial ring as the constant polynomials.

**Example 10.1.6** $\mathbb{Z}[X] \to \mathbb{Z}[X,Y]$ *(*where $\mathbb{Z}[X,Y] = \mathbb{Z}[X][Y]$*)*

This is the same as the previous example, $[Y]$ is acting as a morphism $\mathbb{Z}[X] \to \mathbb{Z}[X][Y]$.

**Example 10.1.7** $\mathbb{Z}[X] \to \mathbb{Z}[X,Y]$ *(*where $\mathbb{Z}[Y,Z] = \mathbb{Z}[Y][X]$*)*

[3] Actually, defined in a `Category` to which `Fraction` belongs.

Axiom knows that $\mathbb{Z} \to \mathbb{Z}[Y]$ is a coercion, and that the map
$[X] : \mathbb{Z}[Y] \to \mathbb{Z}[Y][X]$ (i.e. `UnivariatePolynomial`) lifts this coercion.

**Example 10.1.8** $\mathbb{Z} \to \mathbb{Q}[X, Y]$

This is built by the chain of coercions

$$\mathbb{Z} \to \mathbb{Q} \to \mathbb{Q}[X] \to \mathbb{Q}[X, Y]$$

**Example 10.1.9** $\mathbb{Z}[X] \to \mathbb{Z}(X)$

This uses the coercion from `Fraction` discussed above, since $[Z](X)$ is represented by `Fraction(UnivariatePolynomial(X,Integer))`

**Example 10.1.10** $\mathbb{Z} \to \mathbb{Z}_5$

This uses the function

$$\text{coerce} : \mathbb{Z} \to \%$$

defined in `IntegerMod(n:PositiveInteger)`

**Example 10.1.11** $\mathbb{Z}_{25} \to \mathbb{Z}_5$

Axiom cannot coerce from `IntegerMod(25)` to `IntegerMod(5)`.

#### Structural coercions

The structural coercions are "lifts" of other coercions. In the case of `Lists`, one may perform a structural coercion as follows:

```
coerce(x:List(A)):List(B) == map(coerce,x)@List(B)

map(f:List(A) -> List(B),x:List(A)):List(B) ==
  nullp(x) => nil()$List(B)
  cons(f(car(x)),map(f,cdr(x)))
```

In the case of `Polynomials`, a similar `map` function exists which uses + instead of `cons`

In the case of `Matrix` a similar map function can be used (though since matrices are of fixed sizes – unlike sets, lists, and polynomials – a default valued matrix (0) has to be created, then all the values substituted into (added to) it).

### 10.1.12 Mathematical solution overview

The solution to the questions raised in section 10.1.2 rely on abstract datatypes. These "collect" large numbers of types into collections of similar types. We will be considering abstract datatypes in the language of universal algebra; specifically, order-sorted algebra (section 10.4.16).

Universal algebra has close links with the ideas of category theory. Indeed, (see chapter 10.3 for category theory and section 10.5.22 for the links between the two theories). Axiom uses the language of category theory to describe its algebraic design. Thus we will also discuss the fundamentals of category theory in relation to this work.

Using the order-sorted algebra framework we will show that there is indeed a strict mathematical definition (definition 10.5.25) which we may use to tell us which type changes are natural, and therefore, in our definition, coercions.

Moreover we will show that this approach allows us to create such coercions algorithmically. (algorithm 10.7.11)

In section 10.1.13 we will show an analogy with moving buildings to the solution of changing the type of (i.e. coercing) something.

### 10.1.13 Constructing coercions algorithmically

Consider the following analogy.

Suppose you have been given the job of moving London Bridge from one location to another. The bridge is too large to move in one piece.

You will merely be supervising the work and will not have to carry any of the bricks yourself, hence efficiency is not your concern.

You are also planning to move other different bridges in the future, and once you have a method for moving one bridge, you would like to be able to apply that method to the next. This means that next time a bridge needs moving, you won't have to do much work at all.

Lucky for you someone has already worked out a way of moving any of the bricks at the very bottom of the bridge to their new location, regardless of any obstruction, which would normally leave civil engineers crying. This person was clearly a magician.

So here is your algorithm for moving London bridge, and hence any future bridge.

```
If the original bridge has no bricks left then finished.
Else,
  take a brick from the top of the original bridge.
  If this brick is a bottom brick then,
    use the magical bottom brick mover, to place
    the brick in the corresponding location of
    the new bridge.
  Else
    hold the brick in place, in the corresponding
    location in the new bridge.
  Endif.
Endif.
Repeat.
```

Indeed, you realize that this method will work for other brick-based constructions, or indeed anything that is "built". (Obviously, there are other considerations, is the new location for the bridge a "natural" one? For example, is it long or tall enough?)

The analogue of the bridge is a "thing" or "item" in our computer algebra system. The original location of the bridge is the original type of the "item" and the new location for the bridge is the analogue of the type to which we are coercing the "item".

Thus what we need to know are:

1. what are the foundations of the bridge?
   (**Analogue**: what are the constants of our type?)
2. what is the length, width, strength of the bridge?
   (**Analogue**: what are the parameters of the type?)
3. how do you alter the bricks to be of a new shape/material?
   (**Analogue**: we need to be able to recursively call the coercion algorithm on types

which are "recursively required for coercion". That is types which form the bits of the "item" once we have chopped it up.)

This is our analogy and we may consider many data types to be constructed (or built) similarly. For example, lists (in the Lisp-sense) are always constructed either by being the empty list, (), or by being the `cons` of something to the front of another list.

As a more complicated example, polynomials are constructed via one of the following mechanisms:

1. by being a member of (the included monomorphic copy of) the underlying ring (a constant polynomial)
2. by being a symbol exponentiated to some positive power (a univariate monic monomial)
3. by being the product of a monic monomial and a univariate monic monomial (a monic monomial)
4. by being the product of a monic monomial and a member of the underlying ring (a monomial)
5. by adding a monomial to a polynomial

So we see that at least two of the most basic types in computer algebra are constructed in this way.

## 10.2 Types in Computer Algebra

### 10.2.1 Introduction

Axiom's main view of things is that every object has a type, and that there are ostensibly four layers.

$$\text{items} \in \text{Domains} \in \text{Categories} \in \text{Category}$$

For example,

$$3 \in \mathit{Integer} \in \mathit{Ring} \in \mathit{Category}$$

and the top layer is the unique distinguished symbol, "Category". In general, this top layer is never referred to, and the three lower layers are all that are ever considered.

Strictly speaking, if one considers Axiom's `Categories` to be categories, and Axiom's `Category` symbol, to signify the category of all the `Categories`[4], then one should write,

$$3 \in \text{Integer}, \quad \text{Integer} \in \text{Obj(Ring)}, \quad \text{Ring} \in \text{Obj(Category)}$$

where $3 \in$ Integer means 3 is an element of the carrier of the principal sort of the signature to which Integer belongs.

Users may extend the Axiom system by writing new `Domains` and `Categories` in the Axiom language called Spad.

A category is a collection of domains all of which are "similar". In [Dave90] the authors state that they designed the Scratchpad system of categories or "abstract algebras"[5] for the following reasons:

[4] `Category` is not one of the `Categories` in Axiom

[5] Axiom's `Categories` can also be thought of as order-sorted algebras, but the order on sorts is never explicitly defined and there are some difficulties in the definition of the operator symbols.

1. economy of effort (section 10.2.1)
2. interest (section 10.2.1)
3. functoriality (section 10.2.1)

### Economy of effort

The most obvious reason for this is the view of inheritance. A category (in Axiom) is said to extend another if it inherits all the other's functions, attributes, and equations. This allows for a significant amount of re-use.

A more important issue in an algebra system is that if one can prove a theorem in some generality, i.e. for all the objects in a category, then one does not have to go around proving the theorem for each object of the category.

### Interest

Some categories are interesting and some are not. For example, `Ring` and `Field` are particularly interesting. Many theorems can be proved *for all* `Fields`.

However (to borrow an example from [Dave90]), the category of all rings which when viewed as Abelian groups, have an involution with precisely one fixed point, is not particularly interesting.

One may think of the designer's concept of "interest" as congruent to "useful" (usually to the user, but occasionally to the designers).

### Functoriality

This is called "higher order Polymorphism" in [Crol93]. There are operators which given objects of a category can create new objects of a given (potentially different) category. For example, `List` is a functor from `SetCategory` to `ListAggregate`

$$\begin{array}{rcl} \text{List} : \text{SetCategory} & \rightarrow & \text{ListAggregate} \\ \text{S} & \mapsto & \text{List(S)} \end{array}$$

Indeed, this target category may even be the distinguished symbol `Category`.

In Axiom, the `Category` constructors (often called functors) are functors from

$$\prod_{n \in \mathbb{N} \cup 0} \text{Category} \rightarrow \text{Category}$$

For example, the `Ring` functor takes no arguments and returns the `Category` of `Ring`.

$$\begin{array}{rcl} \texttt{Ring} : () & \rightarrow & \text{Category} \\ () & \mapsto & \texttt{Ring} \end{array}$$

The special `Category` creation operation, `Join` is also a functor. `Join` is used to declare a new `Category` to be a subcategory of the intersection of two or more `Categories`. Here we illustrate the case of `Join` acting on two `Categories`.

$$\begin{array}{rcl} \text{Join} : (\text{Category}, \text{Category}) & \rightarrow & \text{Category} \\ (\text{A}, \text{B}) & \mapsto & \text{A} \cap \text{B} \end{array}$$

More commonly in Axiom, and Axiom's own use of the word "*functor*" covers cases as follows. This is similar to the case of `Ring`, only less trivial. `ListAggregate` in the `Category` of all types of finite lists.

$$\begin{array}{rcl} \text{ListAggregate : (SetCategory)} & \rightarrow & \text{Category} \\ \text{S} & \mapsto & \text{ListAggregate(S)} \end{array}$$

Here we see the subcategory ListAggregate(S) of the `Category` `ListAggregate`. When using Axiom, one usually thinks of one's type as belonging to the smaller, concrete (sub)Category, ListAggregate(S). Mathematically, one usually thinks of one's type as being in the larger, more abstract `ListAggregate`.

## 10.3 Category Theory

As we have already hinted at, category theory is one of the main foundations of Axiom. In this chapter, we will give the necessary definitions to investigate this claim. We will also investigate the claim in itself.

The definitive text on Category Theory is Mac Lane [Macl91]. The amount of theory we require is relative small. We define categories without worrying about Russell's Paradox [Mend87].

### 10.3.1 About Category Theory

**Definition 10.3.2** *A category, C, consists of two collections. The first collection is called the* `objects` *of the category, or* Obj*(C). The second collection is called the arrows of category, or* Arr*(c).*

*Also, for each arrow, f, there exists two special objects with which it is associated. The first is the source of f, called* `source`*(f). The second is called the target of f, called* `target`*(f).*

*There also exists a "*law of composition*" for arrows:*

$$\begin{array}{l} (\forall g, f \text{arrows})((\text{source}(g) = \text{target}(f)) \Rightarrow \\ (\exists g \circ f \text{arrow})((\text{source}(g \circ f) = \text{source}(f)) \wedge (\text{target}(g \circ f) = \text{target}(g)))) \end{array}$$

*Next, for each object, c, there exists a unique arrow, called the* identity arrow *on c, or* $\text{id}_c$.

*Finally the following two axioms must hold:*

$$\begin{array}{c} (\forall k, g, f \text{ arrows})((g \circ f, k \circ g \text{ arrows}) \Rightarrow \\ ((k \circ (g \circ f), (k \circ g) \circ f \text{ arrows}) \wedge \\ (k \circ (g \circ f) = (k \circ g) \circ f))) \end{array}$$

$$(\forall f \text{ arrows})((\text{id}_{\text{target}(f)} \circ f = f) \wedge (f \circ \text{id}_{\text{source}(f)} = f))$$

To introduce the concept, here are some simple finite categories.

**Example 10.3.3 0** *is the empty category, it has no objects and no arrows*

**Example 10.3.4 1** *is the category with one object and one arrow.*

**Example 10.3.5 2** *is the category with two objects,* $a, b$ *and one non-identity arrow,* $f : a \to b$

**Example 10.3.6** $\downarrow\downarrow$ *is the category with two objects,* $a, b$ *and two non-identity arrows,* $f, g : a \to b$

Now let us consider some more useful categories. The collection of objects in the following examples do not always form a set [Macl91][Bern91][Devl79][Fran73]

**Example 10.3.7 Set** *is the category which has as objects, all sets, and has as arrows, all total functions between sets.*

**Example 10.3.8 Grp** *is the category which has as objects, all groups, and has as arrows, all group homomorphisms between them.*

**Example 10.3.9 Ring** *is the category which has as objects, all rings, and has as arrows, all ring homomorphisms between them.*

**Example 10.3.10 Poly** *is the following category. An object of* **Poly** *is a set of all polynomials with: variables from a fixed ordered set* $V$*; coefficients from a fixed ring* $R$*; and the exponents of the variables from an ordered free monoid with one generator* $E$*.*

An arrow of **Poly** is a Polynomial homomorphism. That is, a Ring homomorphism which also acts homomorphically on a certain set of functions which act on polynomials.

Some texts call the arrows of a category the *morphisms* of a category. Examples Set, Grp, Ring, and Poly show us that for some naturally occuring categories, the arrows (or morphisms) are just the homomorphisms of the category. Indeed, an arrow of **Set** (a total function) is really just a homomorphism between sets.

So we see that the arrows preserve a certain set of properties for each element of the object.

We also see that every object of **Grp** is an object of **Set**; every object of **Ring** is an object of **Grp**; and every object of **Poly** is an object of **Ring**.

Now similarly, replace the word "object" with the word "arrow" in the previous paragraph and it still holds true.

Hence we see that in an algebra system, there is a certain amount of "inheritance" amongst the categories. The categories are the so-called *abstract datatypes* since they type the usual datatypes.

We can also see how the categories can collect together all similar types. This functionality can be used for the three design goals of Axiom: economy of effort – the code for many similar types need only be written once; interest – collecting similar types together gives the user an identiical interface to similar types; and functoriality which is best left to be discussed after the following definition.

**Definition 10.3.11** *Let* $B$*,* $C$ *be categories, then a functor,* $T : C \to B$ *consists of two functions*

1. $T : \mathrm{Obj}(C) \to \mathrm{Obj}(B)$ *(the* object *function)*
2. $T : \mathrm{Arr}C \to \mathrm{Arr}B$ *(the* arrow *function)*

*These must obey the following:*

$$\begin{array}{rcl}(\forall f \text{ arrow})(T(\text{source}(f)) & = & \text{source}(T(f)));\\ (\forall f \text{ arrow})(T(\text{target}(f)) & = & \text{target}(T(f)));\\ (\forall c \text{ object}) & = & (\text{id}_{T(c)})\\ (\forall g, f \text{ arrows})((\text{source}(g) = \text{target}(f)) & \Rightarrow & (T(g \circ f) = T(g) \circ T(f)))\end{array}$$

This is a very useful definition. It shows us that when we have two objects and an arrow between them in one category, then a "sensible" map of these objects to another category will induce the obvious map between these image objects.

In fact, one can see that if one were to define a category which has as objects "all categories", and as arrows "all functors" then we would (set theoretical concerns aside) have a well-defined category.

Categories, just like many other mathematical constructs, may form products.

**Definition 10.3.12** *For two categories $B$ and $C$ we may construct a new category denoted $B \times C$ called the product of $B$ and $C$.*

*An object of $B \times C$ is a pair $\langle b, c\rangle$ where $b$ is an object of $B$ and $c$ an object of $C$*

*An arrow of $B \times C$ is a pair $\langle f, g\rangle$ where $f : b \to b'$ is an arrow of $B$, $g : c \to c'$ is an arrow of $C$, the source of $\langle f, g\rangle$ is $\langle b, c\rangle$ and the target of $\langle f, g\rangle$ is $\langle b', c'\rangle$.*

*Composition of arrows is $\langle f', g'\rangle : \langle b', c'\rangle \to \langle b'', c''\rangle$ and $\langle f, g\rangle : \langle b, c\rangle \to \langle b', c'\rangle$ is defined via*

$$\langle f', g'\rangle \circ \langle f \circ g\rangle = \langle f' \circ f, g' \circ g\rangle$$

Axiom makes use of such products implicitly in its functor definitions. For example, see the discussion of `PolynomialCategory` below.

"Natural transformations" are another important part of category theory. They are to functors as functors are to categories. Here is a more formal definition.

**Definition 10.3.13** *For two functors $S, T : C \to B$ a natural transformation $\tau : S \dot{\to} T$ is a function which assigns to every $c \in \text{Obj}(C)$ an arrow $\tau_c = \tau c : Sc \to Tc$ of $B$ such that*

$$(\forall c \ \in \text{Obj}(C))(\forall f : c \to c' \in \text{Arr}(C))(Tf \circ \tau c = \tau c' \circ Sf)$$

Now we may define the following interesting category.

**Definition 10.3.14** *For two categories, $B, C$, the* functor category *$B^C$ is the category with objects the functors $T : C \to B$ and arrows, the natural transformations between two such functors.*

As another fairly abstract definition, consider the following.

**Definition 10.3.15** *Let $S : D \to C$ be a functor and $c \in \text{Obj}(C)$. A universal arrow $c \to S$ is a pair $\langle r, u\rangle \in \text{Obj}(D) \times \text{Arr}(C)$ where $u : c \to Sr$ such that*

$$(\forall (d, f) \in \text{Obj}(D) \times \text{Arr}(C)\text{where} f : c \to Sd)(\exists! f' : r \to d \in \text{Arr}(D))(Sf' \circ \ u = f)$$

Functoriality in a computer algebra system allows us to view objects of one category as objects of another.

As we have already seen, an object of `Poly` is an object of `Ring` and hence an object of `Grp` and thus an object of `Set`. We have also seen this is true for their arrows. This relationship is called the subcategory relationship, defined formally as follows.

**Definition 10.3.16** *A category $C$ is a subcategory of a category $B$ iff every object of $C$ is an object of $B$ and every arrow of $C$ is an arrow of $B$.*

Functors from subcategories to the categories of which they are subcategories are often called "forgetful functors". This is more often true when the target of the functor is *Set*.

Axiom's designers also use functors to create instances of abstract datatypes. `PolynomialCategory(.,.,.)` (Axiom's equivalent of `Poly`) is in Axiom's view a functor.

$$\begin{gathered}\text{Ring} \times \text{OrderedAbelianMonoid} \times \text{OrderedSet} \to \text{PolynomialCategory}(.,.,.) \\ (\text{R}, \text{E}, \text{V}) \mapsto \text{PolynomialCategory}(\text{R}, \text{E}, \text{V})\end{gathered}$$

This functoriality provides the "glue" for Axiom's type mechanism.

Now, trivially for categories $A, B$ if $\exists F : A \to B$ a forgetful functor, the $B$ is a subcategory of $A$. Equally trivially, a concrete instance of PolynomialCategory(R,E,) is a subcategory of `PolynomialCategory(R,.,.)` which is a subcategory of Poly.

It is this first form of subcategory relation that provides Axiom's inheritance mechanism.

### 10.3.17 Categories and Axiom

**Notation 10.3.18** *To distinguish between Axiom's internal structures and those commonly used in mathematics, things which belong to Axiom will be writtin in* **this font**. *Specifically,*

- **Category** *will always refer to Axiom's distinguished symbol, to which all Axiom's* **Categories** *belong.*
- *Hence, a* **Category** *is an Axiom object declared to be such an object, e.g.* **Ring**, **PolynomialCategory(R,E,V)**
- *A* **Domain** *is an Axiom object declared to be a member of a particular* **Category**, *e.g.* **Integer**, **Polynomial(Integer)**
- *An* **item** *is an element of a* **Domain**, *e.g. 1, 5*x**2+1*

Axiom's main view of things is that every object has a type, and that there are four layers.

$$\textbf{items} \in \textbf{Domains} \in \textbf{Categories} \in \textbf{Category}$$

**Categories** also may inherit from or extend other **Categories**, forming an inheritance lattice. Thanks to the higher order polymorphism available in Axiom, **Categories** may also be parameterized by **items**[6] or **Domains**.

This parameterisation may cause some confusion. For example,

$$\textbf{List}(\textbf{S}) : \textbf{ListAggregate}(\textbf{S})$$

declares for each and every **S**, **List(S)** is in the category **ListAggregate(S)**. For example, **List(Integer)** is an object of **ListAggregate(Integer)**. However, **ListAggregate(Integer)** is a mere subcategory of "the category of all objects which are domains of linked lists". This is **ListAggregate(S)**.

[6] One may view a domain as a category, whose objects are the items, and whose arrows are either trivial (i.e. solely the identity arrows) or some other natural occurring meaning, e.g. in **PositiveInteger** one may think that there is a natural map $35 \to 5$, since $5|35$. This would then mean that this map could be "lifted" to **IntegerMod 35** $\to$ **IntegerMod 5**

So **ListAggregate(S)** contains, for example, both **List(Integer)** and **List(Fraction(Integer))** as objects, but what are the arrows of this category? This is something which is not made explicit in the Axiom literature, and is indeed the core of this work.

We will discuss what it means to be a "natural map" and how this related to categorical arrows.

### 10.3.19 Functors and Axiom

Axiom describes its domain constructors as functors, and this is true. After all, (neglecting difficulties with constructors which take domain elements for arguments) these constructors are maps from a cross product of categories to another category.

The difficulties with constructors which take domain elements for arguments disappear when considering the argument use in the footnote in section 10.3.17.

### 10.3.20 Coercion and category theory

If **coerce : A** $\rightarrow$ **B**, then we are going to have that **A** and **B** are objects of the same category **ACAT**, and that **coerce** is an arrow of that category. (See definition 10.5.25)

Also, if **T** is a functor from **ACat** to **TCat**, which in Axiom would look like

$$\mathbf{T}(\mathbf{A} : \mathbf{ACat}) : \mathbf{TCat}$$

then **T** lifts the arrows of **ACat** to **TCat**, and in particular

$$\mathbf{T} : \mathbf{coerce} : \mathbf{A} \rightarrow \mathbf{B} \mapsto \mathbf{coerce} : \mathbf{TCat}(\mathbf{A}) \rightarrow \mathbf{TCat}(\mathbf{B})$$

In many types (and some other languages) **T** is acting like the familiar "map" operator on the **coerce** function.

### 10.3.21 Conclusion

We have seen in this section how category theory and abstract datatyping especially with respect to Axiom's **Category** mechanism are ideologically similar.

We have also shown how Axiom's functors interact with coercions from a category theoretical perspective.

## 10.4 Order sorted algebra

In this section we will introduce the concepts of universal algebra. We will look at the unsorted case to start with and then move on to the sorted case.

We then follow up with the equational calculus which allows us to consider sets of equations which must hold in a concrete instance of an algebra.

All the work in this section is taken from [Dave93a] except for a couple examples.

### 10.4.1 Universal Algebra

Universal algebra will provide us with a natural way of representing categories of types which possess certain functions. In the following lengthy section of definitions, keep in mind that what we will define as a "signature" will be equivalent (in some sense) to our notion of a category (or abstract datatype). An algebra will be the ideological equivalent of an object (or type).

**Definition 10.4.2** *A sort-list $S$, is a (finite) sequence of symbols (called sorts) normally denoted $(s_1, \ldots, s_m)$.*

**Definition 10.4.3** *Given a sort-list of size $m$, a set $A$ of $S$-carriers is an ordered $m$-tuple of sets $A_{s_i}$, indexed by $S$.*

**Example 10.4.4** *One may consider the sort-list of a vector space to be $(K,V)$ where $K$ is to be identified with the underlying field and $V$ is to be identified with the set of all points in the vector space. Then considering $\mathbf{C}$ to be a vector space of $\mathbf{R}$, the $(K,V)$-carriers of $\mathbf{C}$ are ($\mathbf{R}$, $\mathbf{C}$).*

So we see that the carriers of a particular type are a list of types which "have something to do with" the type in question. The sort list is a list of the same length where the $i$th element is the symbol corresponding to a particular abstract datatype to which the $i$th carrier belongs[7].

**Definition 10.4.5** *Given a sort-list $S$ and $n \in \mathbb{N} \cup \{0\}$, an $S$-arity of rank $n$ is an ordered $n$-tuple of elements of $S$.*

An arity is merely a list of elements of the sort list. This will be useful when we wish to type polymorphic (or abstract) functions.

**Definition 10.4.6** *Given a sort-list $S$, $n \in \mathbb{N} \cup \{0\}$, $q = (q_1, \ldots, q_n)$ an $S$-arity of rank $n$, and a set $A$ of $S$-carriers, we define*

$$A^q := \prod_{i \in \{1,\ldots,n\}} A_{q_i}$$

This is the map of an arity to the list of carriers.

**Definition 10.4.7** *An $S$-operator of arity $q$ is a function from $A^q$ to one of the elements of $A$.*

**Definition 10.4.8** *An $S$-operator set or $S$-sorted signature is a set $\Sigma$ of sets $\Sigma_{n,q,s}$ indexed by $n \in \mathbb{N} \cup \{0\}$, $q$ an $S$-arity of rank $n$, and $s \in S$, such that $\cup_{n,q,s}\Sigma_{n,q,s}$ is a subset of some alphabet. An element of some $\Sigma_{n,q,s}$ is called an operator symbol.*

*The usual notation for such a signature is $(\Sigma,S)$.*

So this defines a set of sets of polymorphic functions[8] for a particular sort-list. A signature is like a category in that it is an abstract datatype.

A signature collects together all types which share a similar family of operators.

**Example 10.4.9** *Monoids: $\mathbb{N} \cup \{0\}$ is an additive monoid, whereas $\mathbb{N}$ is a multiplicative monoid.*

[7] Notice that the sort list is defined first and that the carriers depend on the sort list. One does not define a list of carriers and then fix a list of abstract datatypes *post facto*.

[8] Functions without methods.

*The signature for monoids could be viewed as being*

$$\langle \Upsilon, U \rangle = \langle (M, B), ((c), (), (id), (ep), (), (), (b), () \ldots) \rangle$$

*where $M$ is the monoid sort, $B$ is the sort of boolean logic types. The operator symbol $e$ is a member of $\Upsilon_{(0,(),M)}$ and corresponds to the function which always returns the identity constant*[9] *of the monoid.*

*id is the operator symbol in $\Upsilon_{(1,(M),M)}$ which corresponds to the identity function.*

*ep is the operator symbol in $\Upsilon_{(1,(M),B)}$ which corresponds to the "is this the identity element?" function.*

*Last (in our example) but by no means least, b is the monoid's binary operator from $\Upsilon_{(1,(M,M),M)}$.*

*So in $\mathbb{N} \cup \{0\}$, e, id, ep, b correspond to 0, id, 0?, +, respectively. Whereas in $\mathbb{N}$, e, id, ep, b, correspond to 1, id, 1?, $\times$ respectively.*

**Definition 10.4.10** *A (multi-sorted, total) $\Sigma$-algebra is an ordered pair $\langle A, \alpha \rangle$ where $A$ is an $S$-carrier set and*

$$\alpha = \{\alpha_{n,q,s} | n \in \mathbb{N} \cup \{0\}, q \text{ an arity of rank } n, s \in S\}$$

$$\alpha_{n,q,s} = \{\alpha_{n,q,s,\sigma} : A^q \to A_s\}_{\sigma \in \Sigma_{n,q,s}}$$

*So if $\langle \Upsilon, U \rangle$ is the monoidal signature, the*

$$\langle (\mathbf{N}, \mathbf{Boolean}), ((1), (), (id), (1?), (), (), (\times), \ldots) \rangle$$

*is an $\Upsilon$-algebra.*

**Notation 10.4.11** *Let $\langle A, \alpha \rangle$ be a $\Sigma$-algebra. For an operator symbol $\sigma_{n,q,s}$ of the signature $\langle \Sigma, S \rangle$ the function associated with this symbol in $\langle A, \alpha \rangle$ is represented by either $\alpha_{n,q,s,\sigma}$ or $\alpha_{\sigma_{n,q,s}}$.*

This second form of notation is useful for when we refer to operator symbols by names other than those in the form $\sigma_{n,q,s}$. For example, if $\tau = \sigma_{n,q,s}$ then

$$\alpha_{n,q,s,\sigma} = \alpha_{\sigma_{n,q,s}} = \alpha_\tau$$

**Definition 10.4.12** *Given $\langle A, \alpha \rangle$, $\langle B, \beta \rangle$ both $\Sigma$-algebras, then a $\Sigma$-homomorphism $\phi : \langle A, \alpha \rangle \to \langle B, \beta \rangle$ is an $S$-indexed family of functions $\phi_s : A_s \to B_s$ (for each $s$ in $S$) such that*

$$(\forall n \in \mathbb{N} \cup \{0\})(\forall q \text{ arities of rank } n)(\forall (a_1, \ldots, a_n) \in A^q)(\forall s \in S)(\forall \sigma \in \Sigma_{n,q,s})$$

*we have*

$$\phi_s(\alpha_{n,q,s,\sigma}(a_1, \ldots, a_n)) = \beta_{n,q,s,\sigma}(\phi_{q_1}(a_1), \ldots, \phi_{q_n}(a_n))$$

As an example of a homomorphism, let us consider our monoidal case once more. The map $\psi$ from $\mathbb{N} \cup \{0\}$ which maps[10] $n \mapsto 2^n$ is a homomorphism.

[9] Constants are often represented by (if not compiled in the same way as) functions in programming languages, such as Axiom's Spad language. This gives a homogeneous interface for providing constants.

[10] We haven't provided an exponentiation function in our monoidal algebra, but adding an extra sort and a function it is possible. Otherwise think of the map as taking 0 to 1; 1 to 2; 2 to 2×2; 3 to 2×2×2; and so on.

### 10.4.13 Term Algebras

Term algebras provide us with at least one example of an algebra for each signature. In some sense, it is the "free-est" algebra of the signature and all other algebras are isomorphic to quotients of the term algebra.

Notice that given $\langle \Upsilon, U \rangle$ the definition of the monoidal signature from Example 10.4.9, the term algebra is not the free monoid, since we have not added in any "laws" or "equations" to the algebra. Thus we do not have associativity in the term algebra, whereas we do in the free monoid.

Thus we see that, as yet, universal algebra does not model real mathematics perfectly. However this situation will be remedied somewhat in section 10.4.28.

**Notation 10.4.14** *We will define the set $\Delta$ to be the set containing three special symbols*

$$\Delta := \{(\} \cup \{)\} \cup \{,\}$$

$\Delta$ is just a piece of notation that will make the following definition less verbose.

**Definition 10.4.15** *Let $X$ be an $S$-indexed family of sets disjoint from each other, from the set $\Delta$ and from $\bigcup_{n,q,s} \Sigma_{n,q,s}$. We define $T_\Sigma(X)$ to be the $S$-indexed family of sets of strings of symbols from $\bigcup_{s\in S} X_x \cup \Delta \cup \bigcup_{n,q,s} \Sigma_{n,q,s}$ each set as small as possible satisfying these conditions:*

1. $(\forall s \in S)(\Sigma_{0,(),s} \subseteq T_\Sigma(X)_s)$
2. $(\forall s \in S)(X_s \subseteq T_\Sigma(X)_s)$
3. $(\forall \sigma \in \Sigma_{n,q,s})(\forall i \in \{1, \ldots, n\})(\forall t_i \in T_\Sigma(X)_{q_i})(\sigma(t_1, \ldots, t_n) \in T_\Sigma(X)_s)$

*We make $T_\Sigma(X)$ into a $\Sigma$-algebra by defining operators $\sigma_T$ on $T_\Sigma(X)$, for each $\sigma \in \Sigma_{n,q,s}$ via:*

- *If $n = 0$ then $\sigma_T := \sigma$ (Guaranteed to be in $T_\Sigma(X)$ by (1)).*
- *Else, define $\sigma_T(t_1, \ldots, t_n)$ to be the string $\sigma(t_1, \ldots, t_n)$*

*$T_\Sigma(X)$ is called the* term algebra*, and an element of $T_\Sigma(X)$ is called a* term*.*

### 10.4.16 Order-sorted algebras

Order sorted algebras extend the concept of universal algebras by imposing an order on the elements of the sort-list. This can be useful if we know that all algebras of our signature $\langle \Sigma, S \rangle$ have the carrier of $S_2$ as a subset of $S_1$, say.

The ordering of sorts imposes a subtype lattice on the sorts (and hence their carriers). This can then be used in any algebra of the signature to either restrict a function already defined on one type to a subtype of that type, or extend a function on a type to a partial functon of the supertype. Partial functions and order sorted algebra are discussed in section 10.5.1.

Firstly, we had better define what we mean by an "order".

**Definition 10.4.17** *A strict partial order on a set $S$ is a relation $\prec$ on $S$ which is transitive, antisymmetric, and irreflexive.*

A weak partial order on a set $S$ is a relation $\preceq$ such that, $a \preceq b \Leftrightarrow (a \prec b) \vee (a = b)$. Such a relation is transitive.

**Notation 10.4.18** *Let $S$ be a set of sort symbols, such that there is a partial order $\prec$ defined on $S$, and a "top" element $u$ of $S$ such that $s \prec u$ for all $s$ in $S$.*

$u$ provides us with a universe in which to work. Equivalently, we could set $u$ to be any $\text{class}_w$ which is "big enough" (The term $\text{class}_w$ is defined later.)

**Definition 10.4.19** *Extend $\preceq$ from $S$ to the $S$-arities of rank $n$ by defining $(s_1, \ldots, s_n) \preceq (t_1, \ldots, t_n)$ iff $(\forall i \in \{1, \ldots, n\})(s_i \preceq t_i)$.*

**Definition 10.4.20** *An order-sorted, total $\Sigma$-algebra is an ordered triple $\langle A, \{A_s : s \in S\}, \alpha\rangle$, where $A$ is a class known as the* universe, *$\{A_s, s \in S\}$ is an $S$-indexed family of subsets of $A$, known as the carriers of the algebra, and $\alpha$ is a set of sets of functions $\alpha_{n,q,s} = \bigcup_{\sigma \in \Sigma_{n,q,s}} \{\alpha_{n,q,s,\sigma} : A^q \to A_s\}$, such that:*

1. $A_u = A$
2. *If $s \preceq s'$ in $S$, then $A_s \subseteq A_{s'}$*
3. *If $\sigma \in \Sigma_{n,q,s} \cap \Sigma_{n,q',s'}$, with $s \preceq s'$ and $q' \preceq q$, then $\alpha_{n,q,s,\sigma}|_{A^{q'}} = \alpha_{n,q',s',\sigma}$*

Strictly speaking, that last condition should be

$$\iota_{A_s \to A_{s'}} \circ \alpha_{n,q,s,\sigma}|_{A^{q'}} = \alpha_{n,q',s',\sigma}$$

(where $\iota_{A_s \to A_{s'}}$ is the inclusion operator $A_s \to A'_s$) since otherwise the target of the left hand side would be $A^s$ and of the right hand side would be $A^{s'}$.

This is typical of the sort of "abuse of notation" that computer systems often have to implement.

The definition of order sortedness ensures some confluence amongst operators on subtypes. However, it does not provide enough for most sensible applications. The following definition ensures a more confluent system.

**Definition 10.4.21** *The order-sorted signature $\Sigma$ is regular if whenever $\tilde{q}$ is an arity and $\sigma \in \Sigma_{n,q,s}$ with $\tilde{q} \preceq q$, there is a least pair $q'$,$s'$ such that $\sigma \in \Sigma_{n,q',s'}$ and $\tilde{q} \preceq q'$ and $s' \preceq s$.*

Now, we will extend the definition of term algebras to the order sorted case.

**Definition 10.4.22** *Let $X$ be an $S$-indexed family of sets disjoint from each other, from the set $\Delta$ and from $\bigcup_{n,q,s} \Sigma_{n,q,s}$. We define $T_\Sigma(X)$ to be the $S$-indexed family of sets of strings of symbols from $\bigcup_{n,q,s} X_s \cup \Delta \cup \bigcup_{n,q,s} \Sigma_{n,q,s}$, each set of small as possible satisfying these conditions:*

1. $(\forall s \in S)(\Sigma_{0,(),s} \subseteq T_\Sigma(X)_s)$
2. $(\forall s \in S)(X_s \subseteq T_\Sigma(X)_s)$
3. $(\forall s, s' \in S)((s \preceq s') \Rightarrow (T_\Sigma(x)_s \subseteq T_\Sigma(X)_{s'}))$
4. $(\forall \sigma \in \Sigma_{n,q,s})(\forall i \in \{1, \ldots, n\})(\forall t_i \in T_\Sigma(X)_{q_i})(\sigma(t_1, \ldots, t_n) \in T_\Sigma(X)_s)$

*We make $T_\Sigma(X)$ into a $\Sigma$-algebra by letting the first component be $T_\Sigma(X)_u$ and defining operators $\sigma_T$ on $T_\Sigma(X)$, for each $\sigma \in \Sigma_{n,q,s}$ via:*

- *If $n = 0$ then $\sigma_T := \sigma$. (Guaranteed to be in $T_\Sigma(X)_s$ by (1))*
- *Else, define $\sigma_T(t_1, \ldots, t_n)$ to be the string $\sigma(t_1, \ldots, t_n)$*

*$T_\Sigma(X)$ is called the* algebra, *and an element of $T_Sigma(X)$ is called a* term.

The following theorem proves the "freeness" of term algebras. That is to say all algebras are isomorphic to a quotient of the term algebra. In this way we see that all algebras, once represented as a term algebra and a set of rewrite rules are easily implementable in a rewrite system.

It also says something about the constructibility of types. That is, if $\bigcup_{n,q,s} \Sigma_{n,q,s}$ is finite then clearly only a finite number of functions in each and every $\Sigma$-algebra construct the whole algebra.

Moreover, suppose in every real algebra which we wish to study, a certain set of equations hold (see section 10.4.28). Then all our algebras are isomorphic to factors of the term algebra factored out by that set of equations.

Then if every element of this free-est factor algebra is equal to one constructed by a (potentially very small, finite) subset of a (potentially infinite) $\bigcup_{n,q,s} \Sigma_{n,q,s}$, we may utilise this to construct elements of our algebra.

In the automated coercion algorithm (section 10.7.5) we utilise a small (but not necessarily minimal) set to construct all (or some) of the elements of one of the sorts of an algebra.

**Theorem 10.4.23 (First universality theorem)** *Let $\langle A, \alpha\rangle$ be any $\Sigma$-algebra, $\theta$ any map ($S$-indexed family of maps) from $X$ into $A$. Then there exists a unique $\Sigma$-homomorphism $\theta^*$ from $T_\Sigma(S)$ to $A$ such that $(\forall s \in S)(\forall x \in X_s)$*

$$\theta^*_s(\iota(x)) = \theta_s(x)$$

The proof may be found in [Dave93a]

### 10.4.24 Extension of signatures

In this section we will formalise what we mean for one algebra to be an extension of another, or more importantly, from our point of view, for one algebra to be a portion of another.

More formally we are saying how the abstract datatypes (or categories or signatures) may inherit from each other. This sometimes corresponds to algebras depending on each other.

First, a piece of notation.

**Notation 10.4.25** *If $S$ and $T$ are two sets of sets both indexed by the same set, $I$, we say that $S \overrightarrow{\subseteq} T$ iff $(\forall i \in I)(S_i \subseteq T_i)$*

This extends the definition of subsets to $n$-tuples of sets.

**Definition 10.4.26** *Let $S$, $S'$ be two sort-lists, such that, as sets of symbols $S \subseteq S'$, and that $(\forall s, t \in S)((s \preceq_S t) \Leftrightarrow (s \preceq_{S'} t))$. Let $\Sigma$ be an $S$-sorted signature and $T$ be a $S'$-sorted signature. If $\Sigma \overrightarrow{\subseteq} T$, we say that $\Sigma$ is a* sub-signature *of $T$ and that $T$ is a super-signature of $\Sigma$.*

Thus we have the obvious definition of sub-signature. As an example, the monoidal signature is a sub-signature of the signature for groups. The notion of sub-signature corresponds directly with that of sub-category.

**Definition 10.4.27** *With $S$, $S'$, $\Sigma$, $T$ as in the previous definition, let $\langle A, \alpha\rangle$ be an $S'$-sorted $T$ algebra. Define $\langle A, \alpha\rangle|_\Sigma$, called $\langle A, \alpha\rangle$ restricted to $\Sigma$, to be the $S$-sorted $\Sigma$-algebra with carriers, those carriers of $\langle A, \alpha\rangle$ which are indexed by sort symbols from $S$, and operators, $n$-ary operators $\alpha_\tau$ of arity $q$, and result sort $s$, for every $\tau \in \Sigma_{n,q,s}$.*

This is applying the forgetful functor (from the category (corresponding to) $T$ to that (corresponding to) $\Sigma$) to $\langle A, \alpha \rangle$.

### 10.4.28 The equational calculus

Again, we borrow heavily from Davenport's lecture notes[Dave93a]. Throughout this section, we assume that $S = \{s_1, \ldots, s_n\}$ is our indexing family of sort symbols, and $\Sigma = \{\Sigma_{n,q,s}\}$ is an $S$-sorted signature.

The equational calculus presented here applies to multi-sorted algebra. The reader will see that it clearly may be extended to the order-sorted case.

The equational calculus allows us to add "equations" to our signatures (and hence, algebras). This will allow us to assert facts about all the algebra in a signature. For example, we may wish to note that one of the binary operators is always associative. Other more complicated expressions are available also.

The equational calculus does not allow us to define everything that we need: only those things that are easily definable as equations.

**Definition 10.4.29** *An $S$-indexed family of relations $R = \{R_{s_1}, \ldots, R_{s_n}\}$ on a $\Sigma$-algebra, $\langle A, \alpha \rangle$ is called a $\Sigma$-congruence if it satisfies the following four families of conditions:*

$$\begin{array}{rcll} a \in A_{s_i} & \Rightarrow & aR_{s_i}a & (R) \\ aR_{s_i}b & \Rightarrow & bR_{s_i}a & (S) \\ aR_{s_i}b \text{ and } bR_{s_i}c & \Rightarrow & aR_{s_i}c & (T) \\ (\forall \sigma \in \Sigma_{n,q,s})(a_1 R_{q_1} b_1, \ldots, a_n R_{q_n} b_n) & \Rightarrow & \sigma(a_1, \ldots, a_n) R_s \sigma(b_1, \ldots, b_n) & (C_\sigma) \end{array}$$

The first three conditions imply that $R_{s_i}$ is an *equivalence relation* on the set $A_{s_i}$, while the conditions $(C)$ explain how $R$ relates to the various operators of $\Sigma$. The operators of $\Sigma_{n,q,s}$ are well-defined on the equivalence classes.

**Definition 10.4.30** *Let $\langle A, \alpha \rangle$ be an $\Sigma$-algebra, and $\equiv$ be a $\Sigma$-congruence on $\langle A, \alpha \rangle$. Define $\langle A, \alpha \rangle / \equiv$ to be the $\Sigma$-algebra $\langle B, \beta \rangle$, where the carrier set $B_{s_i}$ of $B$ is the set of equivalence classes of $A_{s_i}$ under the equivalence relation $\equiv_{s_i}$, and $\beta_\sigma$ is the operator defined by*

$$\beta_\sigma([a_1], [a_2], \ldots, [a_n]) = [\alpha_\sigma(a_1, a_2, \ldots, a_n)]$$

*for every operator symbol $\sigma$ in every $\Sigma_{n,q,s}$.*

This is merely the quotient algebra, and to be sure, the following is a theorem.

**Theorem 10.4.31** *The operators $\beta_\sigma$ in the above definition are well-defined.*

The following definition actually turns out to be very important.

**Definition 10.4.32** *We say that a sort $s$ is* void *in the signature $\Sigma$ if $T_\Sigma(\emptyset)_s = \emptyset$.*

Basically, having a void sort $\tilde{s}$ in the signature means that there are no constants of that type ($\Sigma_{0,(),\tilde{s}} = \emptyset$) and that one of the following is true.

- no operators have $\tilde{s}$ as a return type: $((\forall q, n)(\Sigma_{n,q,s} = \emptyset)$ where $q$ is an arity of rank $n)$
- every operator with $\tilde{s}$ as a return type has an argument whose sort is either void or $\tilde{s}$: $((\forall q, n)(\Sigma_{n,q,s} \neq \emptyset) \Rightarrow (\exists i \in \{1, \ldots, n\})(q_i$ is a void sort or $\tilde{s}))$ where $q$ is an arity of rank $n)$

**Lemma 10.4.33** *If $s$ is not void in the signature $\Sigma$, then in every $\Sigma$-algebra $\langle A, \alpha\rangle$, $A_s \neq \emptyset$.*

Goguen and Meseguer suggest the following rules of deduction for a sound multi-sorted equational calculus.

**Notation 10.4.34** *Let $X$ be an $S$-indexed family of sets of variable symbols, such that each $X_{s_i}$ is disjoint from all the others, from the operator symbols of $\Sigma$, and from any symbols in any particular algebras we may be reasoning over.*

*We will be reasoning frequently with $S$-indexed families of sets, and wishing to perform operations on them. Let $X = \langle X_1, \ldots, X_n\rangle$ and $Y = \langle Y_1, \ldots, Y_n\rangle$ be two $S$-indexed families of sets, and define $X \overrightarrow{\cup} Y$ to be the $S$-indexed family $\langle X_1 \cup Y_1, \ldots, X_n \cup Y_n\rangle$. Similarly, we will write $X \overrightarrow{\subseteq} Y$ to indicate that each component of $X$ is a subset of the corresponding element of $Y$.*

*A final piece of notation is that the indexed family of empty sets will be donoted by $\overrightarrow{\emptyset}$.*

**Definition 10.4.35** *A* multi-sorted equation *for the signature $\Sigma$ consists of a triple $\langle Y, T_1, t_2\rangle$ where $Y \overrightarrow{\subseteq} X$, $t_1$ and $t_2$ are terms from the same carrier set of $T_\Sigma(X)$ (or $T_\Sigma(X \overrightarrow{\cup} A)$ if we are dealing with equations in the particular algebra $\langle A, \alpha\rangle$), and every variable occurring in $t_1$ and $t_2$ occurs in the appropriate memeber of $Y$ : $t_1, t_2 \in T_\Sigma(Y)$. Such equations are written $\forall Y t_1 = t_2$.*

*If $t_1$ and $t_2$ come from the same carrier set of $T_\Sigma(X)$ we say that the equation is $\Sigma$-generic.*

One should read the symbol $\forall Y$ as meaning "for all values of all the variables of $Y$ in the appropriate sorts (and there had better be some values in those sorts)". It is this interpretation that will solve the paradox mentioned earlier.

**Definition 10.4.36** *An* equational system *for the signature $\Sigma$ is a set of equations for $T_\Sigma(X)$ (or $T_\Sigma(X \overrightarrow{\cup} A)$ if we are dealing with equations in a particular algebra $\langle A, \alpha\rangle$).*

We will tend to write $e = f$ for an equation from an equational system, meaning $\forall Y e = f$ where $Y$ is the $S$-indexed family of sets of variables consisting *precisely* of those variables occurring in $e$ and $f$.

**Notation 10.4.37** *$x_i/t_i$ means substitute the variable $x_i$ with the term $t_i$. We call this a* substitution instance.

**Definition 10.4.38** *A* proof in the multi-sorted equational calculus *of the equation $\forall Y e = f$ from the equational system $\mathcal{E}$ is a finite set of equations $\forall Y_i e_i = f_i$ such that each equation is justified by one of the seven following rules of inference.*

$$\overline{\forall} Y \quad e[x_1/t_1, \ldots, x_n/t_n] = f[x_1/t_1, \ldots, x_n/t_n] \quad (E)$$

*where $\forall X e = f$ is an equation of $\mathcal{E}$, the $t_i$ are terms of the appropriate sort of $T_\Sigma(X)$ and $Y$ is the $S$-indexed family of sets whose $s$-th component is the set of all variables of sort $s$ in all the $t_i$ and those variables of $X_s$ which have not been substituted for, i.e. which are not one of the $x_i$.*

$$\overline{\forall} Y \quad e = e \quad (R)$$

*where $Y$ is the $S$-indexed family of sets whose $s$-th component is the set of all variables of*

*sort $s$ in $e$*

$$\forall Y\, e = f \quad \overline{\forall} Y\, f = e \qquad (S)$$

$$\forall Y\, e = f \quad \forall Y'\, f = g \quad \overline{\forall} Y \overrightarrow{\cup} Y'\ e = g \qquad (T)$$

$$\left.\begin{array}{l} \forall Y_1\ e_1 = f_1 \ldots \forall Y_n\ e_n = f_n \\ \overline{\forall} Y_1 \overrightarrow{\cup} \ldots \overrightarrow{\cup} Y_n\ \sigma(e_1, \ldots, e_n) = \sigma(f_1, \ldots, f_n) \end{array}\right\} \quad (C_\sigma)$$

*where $\sigma$ is any symbol of $\Sigma_{n,q,s}$, and each $e_i$ is a term of sort $q_i$*

$$\forall Y\ e = f \quad \overline{\forall} Y'\ e = f \quad (A)$$

*where $Y \overrightarrow{\subseteq} Y'$*

$$\forall Y\ e = f \quad \overline{\forall} \langle Y_1, \ldots, Y_{i-1} \backslash \{y\}, Y_{i+1}, \ldots, Y_n \rangle \quad e = f \quad (Q)$$

*where $y$ does not occur in $e$ or $f$, and the sort $i$ is non-void for $\Sigma$.*

We use the notation $\vdash \forall Y\ e = f$ (or $\vdash_{\mathcal{E}} \forall Y\ e = f$ if we wish to make clear which equational system is being considered) to mean that $e = f$ is provable in the equational system using the above rules of inference.

**Definition 10.4.39** *If $\forall Y\ e = f$ is an $S$-sorted $\Sigma$-equation (call it $E$), and $\langle A, \alpha \rangle$ is a $\Sigma$-algebra, then we say that $\langle A, \alpha \rangle$ satisfies $E$, or that $\langle A, \alpha \rangle$ is a model for $E$, if for all $S$-sorted maps $\theta$ from $Y$ to $A$, $\theta^*(e) =_A \theta^*(f)$, where $\theta^*$ is the map from $T_\Sigma(X \overrightarrow{\cup} A)$ to $\langle A, \alpha \rangle$, whose existence is guaranteed by the theorem 10.4.23. We extend the notation to sets of equations $\mathcal{E}$ by insisting that $\langle A, \alpha \rangle$ be a model for each equation in $\mathcal{E}$.*

**Theorem 10.4.40 (The Soundness Theorem)** *If $\langle A, \alpha \rangle$ is a model for $\mathcal{E}$, and $\vdash_{\mathcal{E}} \forall Y\ e = f$, then $\langle A, \alpha \rangle$ is a model for $\mathcal{E} \cup \{\forall Y\ e = f\}$.*

The proof of the soundness theorem may be found in [Gogu82].

**Definition 10.4.41** *Let $\langle A, \alpha \rangle$ be a $\Sigma$-algebra, and let $\mathcal{E}$ be an equational system for $\langle A, \alpha \rangle$. The congruence induced by $\mathcal{E}$, denoted by $\equiv_{\mathcal{E}}$ on $\langle A, \alpha \rangle$ is defined by $A \equiv_{\mathcal{E}} B$ if, and only if, $\vdash_{\mathcal{E}} \forall Y\ a = b$, where $Y$ is the $S$-sorted family of empty sets.*

The following two definitions are very important. A theory specifies a type, and we define a variety to be the collection of all types which model that theory.

**Definition 10.4.42** *We define a theory to be the ordered pair $\langle \langle \Sigma, S \rangle, S \rangle$ where $\Sigma$ is an $S$-sorted signature and $S$ is a set of $S$-sorted $\Sigma$-equations. We say that a $\Sigma$-algebra* models *or* satisfies *the theory iff it is a model for $S$.*

**Definition 10.4.43** *The collection of all models of a particular theory is called a variety*

Clearly, since a signature $\langle \Sigma, S \rangle$ may be viewed as a theory $(\langle \Sigma, S \rangle, \emptyset)$, the collection of all $\Sigma$-algebras forms a variety.

When referring to the variety of all models of a particular signature, (for example, $\langle \langle \Sigma, S \rangle, S \rangle$) we usually say the "variety defined by (or specified by) the signature $\langle \langle \Sigma, S \rangle, S \rangle$."

**Theorem 10.4.44 (Second universality theorem)** *Let $X$ be an $S$-sorted set of variables, $\Sigma$ an $S$-sorted operator set, $\mathcal{E}$ a set of equations for $\Sigma$, $\langle A, \alpha \rangle$ a $\Sigma$-algebra which is a model for $\mathcal{E}$, and $\theta$ an $S$-sorted mapping from $X$ to $A$. Then there is a unique $\Sigma$-homomorphism $\theta^{**}$ from $T_\Sigma(X)/\equiv_{\mathcal{E}}$ to $A$ such that*

$$\theta^{**}(\iota^*(x)) = \theta(x) \quad (**)$$

*for all $x \in X$, where $\iota^*$ is the map from $X$ into $T_\Sigma(X)/\equiv_{\mathcal{E}}$ defined by $x \mapsto [x]$.*

**Theorem 10.4.45 (The Completeness Theorem)** *If every $\Sigma$-algebra which satisfies the equation $\mathcal{E}$ also satisfies the equation $\forall Y\ e = f$, then $\vdash_{\mathcal{E}} \forall Y\ e = f$.*

Again, the proof of this may be found in [Gogu82]

Finally, we state the definition of an extension and a protecting extension. Extensions are self-explanatory.

**Definition 10.4.46** *Suppose that $\Sigma$ and $\Sigma \overrightarrow{\cup} T$ are signatures where $\Sigma$ is $S$-sorted and $\Sigma \overrightarrow{\cup} T$ is $S'$-sorted where $S \subseteq S'$. Then $\Sigma \overrightarrow{\cup} T$ is said to be an extension of $\Sigma$.*

A protecting extension is an extension which preserves the equational system for the theory. Combining the second universality theorem in section 10.4.44 with the definition of protecting extension allows us to view a model of a theory as a model of a protecting extension of that theory.

More importantly, if we have an algebra which is a model for a protecting extension of a theory, then we may view the algebra as a model of that theory (unextended).

For example, if the Ring theory is defined to be a protecting extension of the Group theory, then we may view any ring (Ring-algebra) as a group (Group-algebra).

**Definition 10.4.47** *With $\Sigma$, $T$, $S$, and $S'$ as in 10.4.46, Let $\mathcal{E}$ be an equational system on $\Sigma$. Also let $\mathcal{E} \cup \mathcal{F}$ be an equational system on $\Sigma \overrightarrow{\cup} T$. Such an extension is called a protecting extension if*

$$T_{\Sigma \overrightarrow{\cup} T}(X)/\equiv_{\mathcal{E} \cup \mathcal{F}} |_{\Sigma}$$

*is isomorphic to $T_{\Sigma}(X)/\equiv_{\mathcal{E}}$ (isomorphism meaning "isomorphism as $\Sigma$-algebras").*

Notice that the definition of protecting extension is equivalent to the notion of *enrichment* given in [Pada80] (although, this does not allow for the existence of $S'$). Thus a theory $\Omega_1$ is a protecting extension of another $\Omega_0$ iff $\Omega_1$ is complete and consistent with respect to $\Omega_0$.

### 10.4.48 Signatures, theories, varieties, and Axiom

Axiom's type system uses the terminology from category theory, yet its design is based on order sorted signatures.

A **Category** definition in Axiom (i.e. the source code that defines the **Category**) is equivalent to a signature or theory, being a specification of a type or types.

The **Category** viewed as a collection of objects (**Domains**) is the variety specified by the theory which defined the **Category** in the **Category** definition.

The sorts are **%** for the (principal) sort (see definition 10.7.2), and the argument and return types of all the operator symbols of the **Category**[11].

For example, **PolynomialCategory(R,E,V)** is a **Category**. **%** is the (principal) sort. Other sorts include **R**, **E**, **V** a sort each for the **Boolean** and **PositiveInteger** types.

The equations of a theory in an Axiom **Category** definition are either defined in the comments or a certain "attributes" of operators. For example **commutative(*)** means that ***** is commutative and the traditional commutativity equation (for the operator *****) is an equation of the theory. (There is currently no method for enforcing the equations to hold in any model.)

[11] Unless they are concrete types – **Domains**. See section 10.9.5

An algebra (or model for a theory) in Axiom is a **Domain**. Declaring a **Domain** to be in a **Category** is equivalent to saying that it is an algebra of that signature (or model of that theory). Or in other words, a member of the variety defined by the theory which specifies the **Category**.

For example, **Polynomial(Integer)** is a model for **PolynomialCategory(R,E,V)**.

An operator symbol in Axiom is a function declaration in a **Category**. An operator name in Axiom always corresponds to the operator symbol.

For example, + is an operator symbol in **Ring**. In **Integer**, a model for **Ring**, the operator name of + is (and has to be) +.

A carrier is the concrete type substituted for a parameter in any instantiation of a **Domain**.

In **Polynomial(Integer)**, **Polynomial(Integer)** is the carrier **%**. Whereas **Integer**, **NonNegativeInteger** and **Symbol** are the carriers of **R**, **E**, and **V**, respectively. (No sort is given for the **Boolean** and **PositiveInteger** types.)

All of Axiom's extensions are protecting extensions. That is, if a **Category** is declared to extend another then it is always a protecting extension of that **Category**.

In section 10.9 we will discuss how Axiom differs from order sorted algebra and how we may remedy this situation.

### 10.4.49 Conclusion

In this section we have introduced all the basics of order sorted algebra and the equational calculus. We have also shown how these notions are represented in Axiom.

We have demonstrated that order sorted algebra combined with the equational calculus provides a sound basis for a computer algebraic type system.

A multi-sorted signature is a specification for a type and hence is an abstract datatype.

Adding an order on the sorts to obtain an order sorted signature enforces relations between the sorts. It also guarantees sensible interaction between these carriers of these sorts and operators thereon in any algebra of the signature.

Combining signatures with the equational calculus yields theories. Theories further enhance the usefulness of signatures by ensuring certain equations will hold in any model (algebra) of the theory (signature).

## 10.5 Extending order sorted algebra

We will discuss how partial functions and conditional signatures may "tie-in" with traditional order sorted algebra. This will mean that any facts that we may prove for or use from traditional order sorted algebra will also apply when partial functions and/or conditional signatures are considered, provided certain extra facts hold.

Next, we will look at the similarities between order sorted algebra and category theory. This will demonstrate why computer algebra systems such as Axiom use category theory terminology, whereas most of this work uses order sorted algebra as its framework.

Last, we define what we mean by a coercion. This definition is fundamental to the rest of this work.

### 10.5.1 Partial Functions

This subsection is based on [Broy88] after a suggestion by Richardson and Martin.

We ask the question, "How do partial functions interact with classical universal algebra?". We need this in case some of the functions used to create our coercions later are only partial. This can happen (see definition 10.7.7). For example, the division operator in any quotient field is partial, but could be viewed as a constructor function.

In our previous section on universal algebra, we defined an $S$-operator of arity $q$, to be a (total) function from some $A^q$ to some element of $A$. (Where $A$ is a set of $S$-carriers.) Is it possible to redefine this so that the $S$-operators are partial? Indeed, is this necessary? One of the original reasons for the "invention" of order-sorted algebra (definition 10.4.20) (rather than multi-sorted algebra (see definition 10.4.10)) was so that all functions could be considered total. See, for example, [Gogu92].

For example, if $\alpha_{\sigma_{1,(P),T}} : A_P \to A_T$ were a partial function in a $\Sigma$-algebra, $\langle A, \alpha \rangle$, then we could insert a new sort $N \prec P$, such that $\alpha_{\sigma_{1,(N),T}} : A_N \to A_T$ were a total function.

In the example of a quotient field, one would introduce a subsort of the integral domain which would represent all the non-zero elements of the domain.

We may proceed by either attempting to redefine all of universal algebra using partial functions, or through some different mechanism. The following mechanism which uses "virtual sorts" turns out to be flawed. Virtual sorts are an on-the-fly way of generating sorts to represent things like the non-zero elements of an integral domain or non-empty stacks.

**Definition 10.5.2** *Let $\langle \Sigma, S \rangle$ define an order-sorted signature. If we define $\langle \Lambda, L \rangle$ to be the signature where*

1. $L = S \cup \{u_1, \ldots, u_m\}$ *(a set of symbols distinct from all those in $S$)*
2. $(\forall i \in \{1, \ldots, m\})(\exists s_{u_i} \in S)(u_i \prec_L s_{u_i})$
3. $(\forall n \in \mathbb{N} \cup \{0\})(\forall s \in S)(\forall q \in S^n)(\Lambda_{n,q,s} = \Sigma_{n,q,s})$
4. $(\exists n \in \mathbb{N})(\exists q \in L^n \backslash S^n)(\exists s \in S)(\Lambda_{n,q,s} \neq \emptyset)$ *(Note that there may be more than one such triple $\langle n, q, s \rangle$)*

*then we say that the $u_i$ are* virtual sorts *of $\langle \Sigma, S \rangle$ and*
*$\lambda_{n,q,s} \in \Lambda_{n,q,s}(n \in \mathbb{N})(q \in L^n \backslash S^n)(s \in S)$ a virtual operator symbol of $\langle \Sigma, S \rangle$.*

Unfortunately, virtual sorts, virtual operators, and homomorphism do not interact in a satisfactory manner. The introductions of a virtual sort in the source of a homomorphism may be meaningless in the target or vice versa.

For traditional examples, like the non-empty stack, they are fine since this has some meaning in all stack-algebras. But for types like "all the elements which do not map to 5 under a particular coercion," then it may be meaningless to create a virtual sort which attempts to represent this.

Broy[Broy88] defined $\Sigma$-algebras as having either partial or total operators. A (partial) $\Sigma$-homomorphism is then defined as follows:

**Definition 10.5.3 ((Partial) homomorphism)** *If $\langle A, \alpha \rangle$ and $\langle B, \beta \rangle$ are $\langle \Sigma, S \rangle$ algebras*[12] *and $\psi$ is a family of partial maps $\psi_s : A_s \to B_s$, the $\psi$ is a (partial) $\Sigma$-homomorpsism $\langle A, \alpha \rangle \to \langle B, \beta \rangle$ iff the following two conditions are fulfilled:*

[12] with partial operators, in our terminology

*Firstly,*

$$(\forall n \in \mathbb{N} \cup \{0\})(\forall q \text{ arities of rank } n)(\forall s \in S)(\forall \sigma \in \Sigma_{n,q,s})$$
$$(\forall (a_1, \ldots, a_n) \in A^q)$$

if both $\psi_s(\alpha_{\sigma_{n,q,s}}(a_1, \ldots, a_n))$ *and* $\beta_{\sigma_{n,q,s}}(\psi_{q_1}(a_1), \ldots, \psi_{q_n}(a_n))$ are defined, then

$$\psi_s(\alpha_{\sigma_{n,q,s}}(a_1, \ldots, a_n)) = \beta_{\sigma_{n,q,s}}(\psi_{q_1}(a_1), \ldots, \psi_{q_n}(a_n))$$

*Secondly.*

$$(\forall n \in \mathbb{N} \cup \{0\})(\forall q \text{ arities of rank } n)(\forall s \in S)(\forall \sigma \in \Sigma_{n,q,s})$$
$$((\forall a_1, a_1' \in A_{q_1}), \ldots, (\forall a_n, a_n' \in A_{q_n})$$
$$(\bigwedge_{i \in \{1,\ldots,n\}} (\phi_q(a_i) =_{\text{strong}} \phi_{q_i}(a_i'))))$$
$$\Rightarrow \phi_s(\alpha_{\sigma_{n,q,s}}(a_1, \ldots, a_n)) =_{\text{strong}} \phi_s(\alpha_{\sigma_{n,q,s}}(a_1', \ldots, a_n'))$$

*Where "*$=_{\text{strong}}$*" is the strong equality defined via*

$$(a =_{\text{strong}}) \Leftrightarrow ((a \text{ defined} \Leftrightarrow b \text{ defined}) \wedge (a \text{ defined} \Rightarrow a = b))$$

Broy's definition of homomorphism (definition 10.5.3) is (as he states) rather liberal. A special factor which he notes is that the composition of partial homomorphisms need not be a homomorphism again.

The following two definitions turn out to be useful in distinguishing between types of partial homomorphism.

**Definition 10.5.4** *Let $\langle \Sigma', S' \rangle$ be a sub-signature of $\langle \Sigma, S \rangle$. A $\Sigma'$-algebra $\langle A, \alpha \rangle$ is said to be a $\Sigma'$-subalgebra of the $\Sigma$-algebra $\langle B, \beta \rangle$ iff*

1. $(\forall s \in S')(A_s = B_s)$
2. $(\forall n \in \mathbb{N} \cup \{0\})(\forall q \text{ arities of rank } n)(\forall s \in S)(\forall \sigma \in \Sigma_{n,q,s})(\alpha_\sigma = \beta_\sigma|_{A^q})$

**Definition 10.5.5** *Let $\langle \Sigma', S' \rangle$ be a sub-signature of $\langle \Sigma, S \rangle$. A $\Sigma'$-algebra $\langle A, \alpha \rangle$ is said to be a weak $\Sigma'$-subalgebra of the $\Sigma$-algebra $\langle B, \beta \rangle$ iff*

1. $(\forall s \in S')(A_s = B_s)$
2. $(\forall n \in \mathbb{N} \cup \{0\})(\forall q \text{ arities of rank } n)(\forall s \in S)(\forall \sigma \in \Sigma_{n,q,s})(\alpha_\sigma \leq_\perp \beta_\sigma|_{A^q})$

*where "*$\leq_\perp$*" is defined via*

$$f \leq_\perp g \Leftrightarrow (\forall x)((f(x) \text{ is not defined } \vee (f(x) = g(x)))$$

Broy notes that any partial homomorphism may be decomposed using the following methodology. A partial homomorphism $\phi : A \to B$ defines a weak subalgebra $A'$ of $A$ defined via

$$(\forall s \in S)(A_s' = \{a \in A_s : \psi(a) \text{ defined}\})$$

with the functions taking the meanings

$$\alpha'_{\sigma_{n,q,s}}(a_1, \ldots, a_n) = \alpha_{\sigma_{n,q,s}}(a_1, \ldots, a_n)$$

if $(\forall i \in \{1, \ldots, n\})(\phi(a_i)$ are defined) and $\phi(\alpha'_{\sigma_{n,q,s}}(a_1, \ldots, a_n))$ is defined. Otherwise $\alpha'_{\sigma_{n,q,s}}(a_1, \ldots, a_n)$ is not defined.

By $\phi'$ we denote the weak partial identity function $A \to A'$. Then we define $\tilde{\phi} : A' \to B$ to be the total homomorphism which is the restriction of $\phi$ to $A'$.

Being total, $\tilde{\phi}$ induces a weak subalgebra $B'$ of $B$ defined via:

$$(\forall s \in S)(B'_s = \{b \in B_s : (\exists a \in A_s)(\tilde{\psi}(a) = b)\})$$

with the functions defined via

$$\beta'_{\sigma_{n,q,s}}(a_1, \ldots, a_n) = \tilde{\phi}(\alpha_{\sigma_{n,q,s}}(a_1, \ldots, a_n))$$

where $b_i = \tilde{\phi}(a_i)$ and $\beta_{\sigma_{n,q,s}}(a_1, \ldots, a_n)$ is defined.

This then induces a total surjective homomorphism $\phi^n : A' \to B'$. Also, by construction $B'$ is a weak subalgebra of $B$. Therefore there also exists a total injective homomorphism $\phi^m : B' \to B$ which is the natural inclusion operator.

As examples consider the following coercions. (We are only looking at the carrier of the principal sort (definition 10.7.2) in these examples.

**Example 10.5.6** $\mathbb{Q} \to \mathbb{Z}_5$

**Fraction Integer → PrimeField 5**

$$A' = \mathbb{Q}\backslash\{q\frac{1}{5} \mid q \in \mathbb{Q}\backslash\{0\}\}$$

$$B' = B$$

Notice that in this example, we are considering a field homomorphism, and that $A'$ is a weak subfield of $A$. (For example / is undefined on the pair (1,5) in $A'$.) Clearly, $B'$ is a subfield of $B$.

**Example 10.5.7** $\mathbb{Q}[x] \to \mathbb{Z}(x) = \mathbb{Q}(x)$

**Polynomial Fraction Integer → Fraction Polynomial Integer**

$$A' = A$$

$$B' = mathbbQ[x] = \{p \mid p \in \mathbb{Q}(x) \wedge (\exists q \in \mathbb{Q}(x))(p = q \wedge \ \text{denom}(q) \in \mathbb{Q})\}$$

In this example, it is an integral domain homomorphism that concerns us. So although $B'$ is only a weak subfield of $B$ this does not matter, as it is a sub-integral domain of $B$.

**Example 10.5.8 Polynomial Fraction Integer → Fraction Polynomial PrimeField 5**

$$A' = \mathbb{Q}[x]\backslash\{\Sigma_{i \in \mathbb{N} \cup \{0\}}(q_i x^i) \in \mathbb{Q}[x] \mid (\exists i \in \mathbb{N} \cup \{0\})(s \in \mathbb{Q}\backslash\{0\})(q_i = s\frac{1}{5})\}$$

$$B' = \mathbb{Z}_5[x] = \{p \mid p \in \mathbb{Z}_5(x) \wedge (\exists q \in \mathbb{Z}_5(x))(p = q \wedge \ \text{denom}(q) \in \mathbb{Z}_5)\}$$

In this example we are again considering an integral domain homomorphism, and both $A'$ and $B'$ are sub-integral domains of $A$ and $B$, respectively.

These show that real examples of coercions may indeed cause $A$ and $A'$ to differ as well as $B$ and $B'$.

As Broy notes, this definition of partial homomorphism allows the everywhere undefined function to be a partial homomorphism. Thus this definition is too weak for our purposes.

**Definition 10.5.9** *A partial homomorphism, $\phi$ is called strict iff*

$$(\forall n \in \mathbb{N} \cup \{0\})(\forall q \text{ arities of rank } n)(\forall s \in S)(\forall \sigma \in \Sigma_{n,q,s})(\forall (a_1, \ldots, a_n) \in A^q)$$

$$\phi(\alpha_\sigma(a_1, \ldots, a_n)) =_{\text{strong}} \beta_\sigma(\phi(a_1), \ldots, \phi(a_n))$$

Broy notes that a strict homomorphism ensures that $B'$ is a $\Sigma$-subalgebra of $B$ (and not just a weak $\Sigma$-subalgebra). $\phi^n$ is always strict, and if $\phi$ is strict, then $B'$ is indeed a $\Sigma$-subalgebra of $B$.

Our requirements for homomorphism are that it act strictly on a certain set of partial functions (the constructors, definition 10.7.7). In other words, we will require that our coercion be a strict partial $\Sigma^C$-homomorphism (again, see definition 10.7.7)

For the rest, we will not in general concern ourselves with the partiality of homomorphisms, operators, or strictness. Since, provided our homomorphisms act strictly on the constructors of the type (and any type recursively required for construction (definition 10.7.18)) then the strong equality in the definition of strictness puts us in good shape.

### 10.5.10 Conditional varieties

Axiom contains the notion of "conditional **Categories**" or in the language of universal algebra, "conditional varieties". In this section we define and reconcile the notions of conditional and non-conditional[13] varieties. This means that in future sections we will be able to ignore the existence of conditional varieties in Axiom.

Notice that our choice of the word conditional here is similar to that used in the phrase "conditional algebraic specifications". The difference being that our conditions are predicates evaluated over arities not terms. Ours is a higher order notion and should not be confused with the ordinary lower order work.

We extend the definition of conditional signature, via the following definition:

**Definition 10.5.11** *A $S$-sorted* conditional signature *is a set $C\Sigma$ of sets $C\Sigma_{n,q,s}$ indexed by $n \in \mathbb{N} \cup \{0\}$, $q$ an $S$-arity of rank $n$, and $s \in S$. Each element of $C\Sigma$ is of the form*

$$\text{if } P(w) \text{ then } \sigma_{n,q,s}$$

*where $P$ is a well-formed proposition in some language, and $w$ is an arity of finite rank (if it is a tautology, it is always represented by $T$). The $\sigma_{n,q,s}$ are the conditional operator symbols.*

Firstly, we need to define a conditional term algebra.

**Definition 10.5.12** *Let $S$ be an $S$-indexed family of sets disjoint from each other, from the set $\Delta$, the set $\{if,P,then\}$ and from*

$$\{\sigma \mid (\exists \text{ arity } w)(\text{“}if\ P(w)\ then\ \sigma'' \in \bigcup_{n,q,s} \Sigma_{n,q,s})\}$$

*We define $T_{C\Sigma}(X)$ to be the $S$-indexed family of sets of strings of symbols from $\{if, P, then\} \cup \bigcup_{s \in S} X_s \cup \Delta \cup \bigcup_{n,q,s} \Sigma_{n,q,s}$ each set as small as possible satisfying these conditions:*

1. $(\forall s \in S)(\Sigma_{0,(),s} \subseteq T_{C\Sigma}(X)_s)$

[13] Unconditional may have been a better choice of word from an English language point of view. However, we are not saying that an algebra is unconditionally an algebra of a variety; we are saying that it is an algebra of a variety which has no conditions

2. $(\forall s \in S)(X_s \subseteq T_{C\Sigma}(X)_s)$
3. $(\forall \text{“}if\ P(w)\ then\ \sigma'' \in C\Sigma_{n,q,s})(\forall i \in \{1,\ldots,n\})(\forall t_i \in T_{C\Sigma}(X)_{q_i})$
   $(\text{“}if\ P(w)\ then\ \sigma''(t_1,\ldots,t_n) \in T_{C\Sigma}(X)_s)$

*We make $T_{C\Sigma}(X)$ into a conditional $C\Sigma$-algebra by defining conditional operators "if $P(w)$ then $\sigma_T$" on $T_{C\Sigma}(X)$, for each "if $P(w)$ then $\sigma'' \in C\Sigma_{n,q,s}$ via:*

- *if $n = 0$ then "if $P(w)$ then $\sigma_T$":="if $P(w)$ then $\sigma$". (Guaranteed to be in $T_{C\Sigma}(X)_s$ by (1)).*
- *Else, define "if $P(w)$ then $\sigma_T(t_1,\ldots,t_n)$" to be the string "if $P(w)$ then $\sigma(t_1,\ldots,t_n)$"*

*$T_{C\Sigma}(X)$ is called the* conditional term algebra*, and an element of $T_{C\Sigma}(X)$ is called a* conditional term*.*

Associated with the notion of conditional signature are the notions of conditional theory and conditional variety. However, we require a new definition for equations and equational systems.

**Definition 10.5.13** *A* conditional (multi-sorted) equation *for the conditional signature $C\Sigma$ consists of a quadruple $(P(w), Y, t_1, t_2)$, where $P$ is a well-formed proposition in some language, and $w$ is an arity of finite rank, (if it is a tautology, it is always represented by $T$) where $Y \overrightarrow{\subseteq} X$, $t_1$ and $t_2$ are terms from the same carrier set of $T_{C\Sigma}(X)$ (or $T_{C\Sigma}(X \overrightarrow{\cup} A)$ if we are dealing with equations in a particular algebra $\langle A, \alpha\rangle$), and every variable occurring in $t_1$ and $t_2$ occurs in the appropriate member of $Y : t_1, t_2 \in T_{C\Sigma}(Y)$. Such equations are more usually written* (if $P(w)$)$(\forall Y\ t_1 = t_2)$*. If $t_1$ and $t_2$ come from the same carrier set of $T_{C\Sigma}(X)$ we say that the equation is $C\Sigma$-generic.*

Note that this definition of conditional equation should not be confused with that found in work such as [Koun90]. In that context, a conditional equation is a clause such as (if $P(t)$)$(\forall Y\ t1 = t2)$ where $t$ is a subterm of $t_1$, for example.

**Definition 10.5.14** *A* conditional equation system *for the signature $\Sigma$ is a set of conditional equations for $T_{C\Sigma}(X)$ (or $T_{C\Sigma}(X) \overrightarrow{\cup} A$ if we are dealing with equations in a particular algebra $\langle A, \alpha\rangle$).*

Associated with definition 10.5.11 are two special signatures $\Sigma$ and $\tilde{\Sigma}$. The former is the extension of all the $C\Sigma$s; the latter is extended by all $C\Sigma$s. More formally,

**Definition 10.5.15** *Using the definitions of definition **??**, we define $\Sigma$ to be the $S$-sorted signature where $\forall n, q, s(n \in \mathbb{N} \cup \{0\}$, $q$ an $S$-arity of rank $n$, and $s \in S)$*

$$\Sigma_{n,q,s} = \{\sigma_{n,q,s} \mid \text{“}\ if\ P(w)\ then\ \sigma_{n,q,s}\ \text{”} \in C\Sigma_{n,q,s}\}$$

*Now similarly, we define $\tilde{\Sigma}$ to be the $\tilde{S}$-sorted signature where $\forall n, q, s(n \in \mathbb{N} \cup \{0\}$, $q$ an $\tilde{S}$-arity rank $n$, and $s \in \tilde{S})$*

$$\tilde{\Sigma}_{n,q,s} = \{\sigma_{n,q,s} \mid \text{“}\ if\ T\ then\ \sigma_{n,q,s}\ \text{”} \in C\Sigma_{n,q,s}\}$$

*where $\tilde{S}$ is the largest subset of $S$ such that $\tilde{\Sigma}$ has no void sorts*

The algebras of conditional signatures are the same as ordinary algebras, except that the proposition $P$ is evaluated over $A^w$, and iff this is true, there is an operator $\alpha_{n,q,s,\sigma}$ in that algebra. More formally,

**Definition 10.5.16** *An* order-sorted, total, conditional $C\Sigma$-algebra *is an ordered triple* $\langle A, \{A_s : s \in S\}, \alpha\rangle$*, where* $A$ *is a class known as the universe,* $\{A_s : s \in S\}$ *is an* $S$*-indexed family of subsets of* $A$*, known as the* carriers *of the algebra, and* $\alpha$ *is a set of sets of functions* $\alpha_{n,q,s} = \bigcup_{\sigma \in \Sigma_{n,q,s}} \{\alpha_{n,q,s,\sigma} : A^q \to A_s \mid P(A^w)\}$*, such that*

1. $A_u = A$
2. *If* $s \preceq s' \in S$*, then* $A_s \subseteq A_{s'}$
3. *If "*$if\ P(w)$ *then* $\sigma'' \in C\Sigma_{n,q,s}$
   *and "*$if\ P'(w')$ *then* $\sigma'' \in C\Sigma n, q', s'$*, with* $s \preceq s'$
   *and* $q' \preceq q$*, and also* $P(A^w)$ *and* $P'(A^{w'})$
   *then* $\alpha_{n,q,s,\sigma} \mid_{A^{q'}} = \alpha_{n,q',s',\sigma}$

Clearly, if all the $P$ are tautologies then a conditional signature $C\Sigma$ is identical to its non-conditional partner, $\Sigma$. Therefore all conditional $C\Sigma$-algebras are (non-conditional) $\Sigma$ algebras in this case.

In any case a $C\Sigma$-algebra is a $\tilde{\Sigma}$-algebra. Thus we can see that $\tilde{\Sigma}$ is a minimal[14] signature for $C\Sigma$.

**Definition 10.5.17** *We define a* conditional theory *to be the ordered pair* $\langle\langle C\Sigma, S\rangle, S\rangle$ *where* $C\Sigma$ *is an* $S$*-sorted conditional signature and* $S$ *is a set of* $S$*-sorted conditional* $\Sigma$*-equations. We say that a* $C\Sigma$*-algebra* $\langle A, \alpha\rangle$ models *or* satisfies *the conditional theory iff it is a model for* $S \mid_{\langle A,\alpha\rangle}$*, where* $S \mid_{\langle A,\alpha\rangle}$ *is defined as follows,*

$$S \mid_{\langle A,\alpha\rangle} := \{(Y, t_1, t_2) \mid ((P(w), Y, t_1, t_2) \in S) \wedge P(A^w)\}$$

A theoretical interpretation of a conditional theory $\langle\langle C\Sigma, S\rangle, S\rangle$ is a theory $\langle\langle \Sigma_\tau, S\rangle, S_\tau\rangle$ equivalent to $\langle\langle C\Sigma, S\rangle, S\rangle$ with all the propositions $P(w)$ evaluated in some consistent manner.

Finally we demand that for all possible pairs of theoretical interpretations $\langle\langle \Sigma_0, S\rangle, S_0\rangle$, $\langle\langle \Sigma_1, S\rangle, S_1\rangle$ where $\Sigma_0 \overrightarrow{\subseteq} \Sigma_1$ and $S_0 \subseteq S_1$ that $\langle\langle \Sigma_1, S\rangle, S_1\rangle$ is a protecting extension of $\langle\langle \Sigma_0, S\rangle, S_0\rangle$.

**Definition 10.5.18** *The collection of all models of a particular conditional theory is called a* conditional variety*.*

In fact the conditional variety defined by $\langle\langle C\Sigma, S\rangle, S\rangle$ is equivalent to the variety defined by $\langle\langle \tilde{\Sigma}, \tilde{S}\rangle, S\rangle$.

Also, the variety specified by any theoretical interpretation of the conditional theory $\langle\langle C\Sigma, S\rangle, S\rangle$ forms a subclass of the conditional variety. Moreover, for a pair of theoretical interpretations $\langle\langle \Sigma_0, S\rangle, S_0\rangle$, $\langle\langle \Sigma_1, S\rangle, S_1\rangle$ where $\Sigma_0 \overrightarrow{\subseteq} \Sigma_1$ and $S_0 \subseteq S_1$, the variety specified by $\langle\langle \Sigma_1, S\rangle, S_1\rangle$ forms a subclass of the variety specified by $\langle\langle \Sigma_0, S\rangle, S_0\rangle$.

If we consider two conditional algebras from the same conditional variety, then we wish to know whether there is a homomorphism from one to the other. For this we need to define "conditional homomorphism".

A technical definition which will allow us to define homomorphisms more easily.

**Definition 10.5.19** *Let* $\langle A, \alpha\rangle$*, and* $\langle B, \beta\rangle$ *be* $C\Sigma$*-algebras. For all* $n, q, s$*, we define*

$$\mu_{\alpha,n,q,s} : \alpha_{n,q,s} \to \Sigma_{n,q,s}$$

[14] Minimal in the sense that it has fewest operator symbols and sorts. The variety it specifies is maximal, if you consider the number of algebras that model it.

*to be the map,*

$$\alpha_{n,q,s,\sigma} \mapsto \sigma_{n,q,s}$$

*Similarly, define* $\mu_{\beta,n,q,s}$ *for* $\langle B, \beta \rangle$.

The family of maps, $\mu_\alpha$ (as we show in notation 10.5.21 map an algebra, $\langle A, \alpha \rangle$ to a non-conditional signature which it models.

**Definition 10.5.20** $\phi : \langle A, \alpha \rangle \to \langle B, \beta \rangle$ *is a* conditional $C\Sigma$-homomorphism *iff it is a non-conditional* $\Sigma^{AB}$*-homomorphism, where*

$$\left( \begin{array}{rcl} \Sigma^{AB} & = & \{ \{\mu_{\alpha,n,q,s}(\alpha_{n,q,s,\sigma}) \mid \alpha_{n,q,s,\sigma} \in \alpha_{n,q,s}\} \cap \\ & & \quad \{\mu_{\beta,n,q,s}(\beta_{n,q,s,\sigma}) \mid \beta_{n,q,s,\sigma} \in \beta_{n,q,s}\} \mid \\ & & \quad n \in \mathbb{N} \cup \{0\}, q \ \tilde{\mathrm{S}} - \text{arity of rank } n, s \in \tilde{S}\} \end{array} \right)$$

$\Sigma^{AB}$ *is a* $\tilde{S}$*-sorted signature where* $\tilde{S}$ *is the largest subset of* $S$ *such that* $\Sigma^{AB}$ *has no void sorts.*

In fact, if we use the following piece of notation.

**Notation 10.5.21** *If* $\langle A, \alpha \rangle$ *is a conditional* $C\Sigma$*-algebra, we define* $\Sigma^{NC(A)}$ *to be the non-conditional signature*

$$\begin{array}{l} \{\{\mu_{\alpha,n,q,s}(\alpha_{n,q,s,\sigma}) \mid \sigma_{n,q,s,\sigma} \in \alpha_{n,q,s}\} \mid \\ \quad n \in \mathbb{N} \cup \{0\}, q \text{ an } \hat{\mathrm{S}} - \text{arity of rank } n, s \in \hat{S}\} \end{array}$$

$\Sigma^{NC(A)}$ *is a* $\hat{S}$*-sorted signature where* $\hat{S}$ *is the largest subset of* $S$ *such that* $\Sigma^{NC(A)}$ *has no void sorts.*

As an aside, notice that for each $C\Sigma$-algebra $\langle A, \alpha \rangle$, there is an associated non-conditional term algebra $T_{\Sigma^{NC(A)}}(X)$. This term algebra is the equivalent of evaluating each $P(w)$ over $A$ in $T_{C\Sigma}(X)$.

Then $\langle A, \alpha \rangle$ is a non-conditional $\Sigma^{NC(A)}$-algebra, and

$$\Sigma^{AB} = \{\Sigma^{NC(A)}_{n,q,s} \cap \Sigma^{NC(B)}_{n,q,s} \mid n \in \mathbb{N} \cup \{0\}, q \text{ an } \breve{\mathrm{S}} - \text{arity of rank } n, s \in \breve{S}\}$$

So, for any two conditional algebras of the same conditional signature, there exists a fixed (maximal) non-conditional signature of which they are both non-conditional algebras. (We may have to forget some operators and even some sorts, which may only exist in conditions).

Our definition of conditional homomorphism is that of the non-conditional homomorphism from this fixed non-conditional signature.

Hence, the concept of conditional signatures can always be reduced to one for non-conditional signatures and therefore any results we prove or use need only be for non-conditional signatures.

As we have seen in definition 10.5.17 this concept may also be extended to theories. In addition to adding new operators when certain propositions hold, we also allow for new equations to be added (conditionally). However, we demand that any such extension is a protecting extension of the minimal non-conditional theory (the theory over the signature $\langle \overline{\Sigma}, \overline{S} \rangle$).

More explicitly, let us define $\hat{S}$ to be all those equations from $S$ which only involve arity from $\Sigma^{NC(A)}$ (as well as all equations not involving arities)[15] Similarly, define $\overline{S}$ to be all those

[15] Alternatively, define $\hat{S}$ to be those equations from $S$ for which $\langle A, \alpha \rangle$ is a model, since it may not model all the equations in the definition of $\hat{S}$ in the main text.

equations from $S$ which only involve arities from $\overline{\Sigma}$ (as well as all equations not involving arities).

If $\langle A, \alpha \rangle$ is a conditional $C\Sigma$-model then $\langle \langle \Sigma^{NC(A)}, \hat{S} \rangle \hat{S} \rangle$ must be a protecting extension of $\langle \langle \overline{\Sigma}, \overline{S} \rangle, \overline{S} \rangle$.

### 10.5.22 A Category theory approach

This section is an aside from the rest of the section. We are not extending the theory of order sorted algebra nor category theory, merely explaining why we cover both.

There is a great deal of correspondence between category theory (specifically, categorical type theory) and universal algebra. There is not enough room here to cover this huge topic, but the reader is pointed to [Crol93], which covers categoriecal type theory in great depth.

As usual, Mac Lane [Macl91] also contains a great deal of information on the correspondence between universal algebra and adjoint functors in the section on "Monads and Algebras".

All the ideas from multi-sorted algebra theory may be represented in the categorical type theory framework, although an equivalent for order-sorted algebra appears not to be covered. However, the author believes that the notion could be introduced.

One area which we will discuss here is the concept of algebraic homomorphism and the arrows (morphisms) of a given category.

The correspondence between a categorical type theory $C$ and its associated algebraic type theory $\langle \langle \Sigma, S \rangle, S \rangle$ is the following

$$\text{Obj}(C) = \text{ all the } \Sigma\text{−algebras for a given } \Sigma \text{ which model } S.$$

$$\text{Arr}(C) = \text{ all (or a fixed sub} - \text{collection of) the } \Sigma - \text{homomorphisms.}$$

The category of all algebras which model a given theory $\langle \langle \Sigma, S \rangle, S \rangle$ has as an initial object the free algebra $T_\Sigma(X)/\equiv_\mathcal{E}$. (This is why the second universality theorem 10.4.44 holds).

So we see that both categories and theories collect together similar types. This abstraction of information forms the basis of abstract datatyping.

### 10.5.23 Coercion

In this work we are interested in coercions which so far have been explained as being natural, type-changing maps. We may define them in a far more strict fashion.

The following would seem to be a good definition for coercion. However, in practice this definition is not well-defined enough since it does not state which theory a coercion should come from.

**Definition 10.5.24** *Let $\langle A, \alpha \rangle$ and $\langle B, \beta \rangle$ be algebras from the variety specified by some theory $\langle \langle \Sigma, S \rangle, S \rangle$. The map $\phi : \langle A, \alpha \rangle \to \langle B, \beta \rangle$ is a coercion if it is a homomorphism of that theory.*

Using the category theory correspondence above, we see that a function between two types of the category of all algebras which model a particular theory is a coercion iff it is an arrow of that category.

The only problem with this definition of coercion is that it does not specify from which theory (or equivalently, category) we should demand the homomorphism be taken. In general, we would like this to be the most specific, or smallest category to which both algebras belong.

If we just used the definition of coercion above (definition 10.5.24) then any total map between types would be a coercion! Any small category may be forgotten back to **Set** via the forgetful functor and any total function is an arrow of **Set**. Most types in a computer algebra system are objects of **Set** (which is equivalent to **SetCategory** in Axiom - the second most basic **Category** in Axiom).

Richardson notes that we also require a fixed framework in which we define our coercions. If we were to attempt to define a coercion $\langle A, \alpha\rangle$ to $\langle B, \beta\rangle$ to be "any map which is a homomorphism for all theories which both algebras model" then the definition would not be well-defined. We need to state a context.

We are attempting to reflect the situation that appears in a system like Axiom. Thus we must state that we have been given *a priori* a fixed collection of theories, and that given any type, we know precisely which theories it models.

This precludes a user from adding another theory which the algebras model which could redefine what "coercion" means for those algebras.

In the category theory correspondence, we say that Axiom's **Category** is a fixed collection of categories, and given any type we know of which categories it is an object.

**Definition 10.5.25 (Coercion)** *Let $\Upsilon$ be a fixed collection of theories.*

*Let $\langle A, \alpha\rangle$ be a model for theories $\Theta_i$ (for $i$ in some indexing set $I$) where $(\forall i \in I)(\Theta_i$ is a theory from $\Upsilon)$.*

*Similarly, let $\langle B, \beta\rangle$ be a model for theories $\Omega_j$ (for $j$ in some indexing set $J$) where $(\forall j \in J)(\Omega_j$ is a theory from $\Upsilon)$.*

*Then we call a map $\langle A, \alpha\rangle$ to $\langle B, \beta\rangle$ a* coercion *iff it is a homomorphism for all the theories in*

$$\{\Theta_i \mid i \in I\} \cap \{\Omega_j \mid j \in J\}$$

In Axiom, **Domains** are only members of a fixed set of **Categories**.[16]

In fact, in Axiom we are in a slightly better position. If a **Domain** is an object of two **Categories** then there must exist a **Category** which extends (potentially trivially) both of these **Categories** of which the **Domain** is an object.

Thus, in Axiom, definition 10.5.25 reduces to,

> "A coercion in Axiom from a type **A** to a type **B** is a homomorphism of the most restrictive **Category** to which both **A** and **B** belong"

It would be useful to have a name for maps which are "coercions" in the sense of definition 10.5.24. These maps are in a sense natural, and may be the next-most natural map between two types. However, as we have shown in the example using **Set** the map need not be particularly "natural" from a realistic point of view.

As it stands, with our current terminology maps which satisfy definition 10.5.24 which are not coercions are merely "homomorphisms which are not coercions".

If no coercion existed between two types, but a "homomorphism which is not a coercion" existed and a user required that homomorphism then either the user only required a conversion

[16] A user could re-implement one of these **Categories** and ruin everything.

or the theory lattice for the algebra system has not been designed correctly.

### 10.5.26 Conclusion

In this section we have shown how classical order sorted algebra may be extended to encompass the notion of partial functions and conditional signatures. We have also stated how these notions interact with the equational calculus. These are important extensions due to the fact that these notions are used extensively in Axiom.

We have also mentioned some of the correlation between category theory and universal algebra. Finally we have made the important definition of a coercion and demonstrated why the definition is necessarily strict.

## 10.6 Coherence

We will look at a conjecture by Weber which is important to our work. We will state his assumptions and proof (which is incorrect).

We will then add in some extra assumptions and truly prove the theorem. Finally we will relax one of Weber's assumptions and prove that the theorem still holds.

The assumptions made by Weber provide a strict formal setting for types in an algebra system. The theorem in itself proves confluence for coercions in this setting.

All the work in this section (except the explanation of "$n$-ary type constructor") and the next are from Weber [Webe93b] and [Webe95]. All the rest is original work.

### 10.6.1 Weber's work I: definitions

This section contains the definitions from Weber's thesis [Webe93b] and [Webe95] required for the statement and Weber's "proof' of Weber's coercion conjecture 10.6.17. His assumptions are in the next section.

These statements will also be used when we correctly prove the coherence theorem 10.6.25 in section 10.6.18

It should be noted that Weber's coercion conjecture only makes up part of one section of his thesis [Webe93b] which also covers various areas of type classes, type inference, and coercion in great depth and detail.

Weber uses the phrase "type class" where we would use the terms "category" or "variety".

**Definition 10.6.2** *A* base type *is any type which is not a parametrically defined type. (i.e. a 0-ary operator in the term algebra of type classes)*

So for example, the **Integer** and **Boolean** types are base types.

**Definition 10.6.3** *A* ground type *is any type within the system which is either a base type or a parametrically defined type with all the paramters present. Any non-ground type is called a* polymorphic type.

As examples, we have **Integer** and **Polynomial(Fraction(Integer))**. As a non-example, we have **Polynomial(R:Ring)**.

**Definition 10.6.4** *If there exists a coercion from $t$ to $t'$ we say that $t \trianglelefteq t'$.*

This definition places an order on the ground types.

Weber uses the phrase "n-ary type constructor" to mean a functor from the product of $n$ categories to a specific category.

Equivalently, it is a function from the product of $n$ varieties (specified respectively by the theories $\Omega_1, \ldots, \Omega_n$) to a variety (specified by the theory $\langle\langle \Sigma, S\rangle, S\rangle$. This is a function which, for all $i$ maps the carrier of the principle sort (and potentially the carriers of some of the non-principal sorts) of a model of $\Omega_i$ to one (for each sort mapped) of the non-principal sorts of a model of $\langle\langle \Sigma, S\rangle, S\rangle$.

If the model returned as $\langle A, \alpha\rangle$, this function must map one and only one sort-carrier to each and every member of $A \backslash \{A_{S_1}\}$ where $S_1$ is the principal sort.

**Definition 10.6.5** *For a ground type $t$ we define* com*$(t)$ to be 1, if $t$ is a base type or if $t = f(t_1, \ldots, t_n)$ (an $n$-ary type constructor) then* com*$(t)$ is defined to be* $1 + \max(\{\mathrm{com}(t_i) \mid \{1, \ldots, n\}\})$.

This defined the "complexity" of a ground type to be how far up the type lattice it is.

**Definition 10.6.6 (Coherence)** *A type system is* coherent *if the following condition is satisfied.*

$$(\forall \textit{ground types } t_1, t_2)((\phi, \psi : t_1 \to t_2 \textit{ coercions}) \Rightarrow (\phi = \psi))$$

This guarantees that there only exist one coercion from one ground type to another. This is a highly desirable feature of any type system. The main results of this section (the coherence theorem 10.6.25 and the extended coherence theorem 10.6.30) are that we may be able to guarantee that our type system is coherent providing some sensible assumptions hold true.

In the following definitions, all the $\sigma$ and $\sigma'$ are type classes.

**Notation 10.6.7** *$t : \sigma$ means that the type $t$ is an object of the type class $\sigma$.*

**Definition 10.6.8** *The $n$-ary type constructor $f$ ($n \in \mathbb{N}$) induces a* structural coercion *if there are sets $\mathcal{A}_f \subseteq \{1, \ldots, n\}$ and $\mathcal{M}_f \subseteq \{1, \ldots, n\}$ such that the following condition is satisfied:*

> *If $f : (\sigma_1, \ldots, \sigma_n) \to \sigma$ and $f : (\sigma'_1, \ldots, \sigma'_n) \to \sigma'$, and $(\forall i \in \{1, \ldots, n\})(\forall$ ground types $t_i : \sigma_i$ and $t'_i : \sigma'_i)(i \notin \mathcal{A}_f \cup \mathcal{M}_f \Rightarrow t_i = t'_i)$ and there exist coercions:*
>
> $$\begin{array}{rll} \phi_i : t_i \to t'_i & \text{if} & i \in \mathcal{M}_f \\ \phi_i : t'_i \to t_i & \text{if} & i \in \mathcal{A}_f \\ \phi_i = \mathrm{id}_{t_i} = \mathrm{id}_{t'_i} & \text{if} & i \neq \mathcal{A}_f \cup \mathcal{M}_f \end{array}$$
>
> *then there exists a* **uniquely defined** *coercion*
>
> $$\mathcal{F}_f(t_1, \ldots, t_n, t'_1, \ldots, t'_n, \phi_1, \ldots, \phi_n) : f(t_1, \ldots, t_n) \to f(t'_1, \ldots, t'_n)$$

*The type constructor $f$ is* covariant *(or* monotonic*) in its $i$-th argument if $i \in \mathcal{M}_f$. $f$ is* contravariant *or (*antimonotonic*) in its $i$-th argument if $i \in \mathcal{A}_f$*

Note that if $i \in \mathcal{A}_f \cap \mathcal{M}_f$ then $t_i \cong t'_i$.

As an example of covariance, the list constructor in Axiom, **List** (a functor **Set** $\to$ **ListAggregate**()) takes one argument in which it is covarient. Given types **A** and **B**,

such that there exists a coercion $\phi_1 : \mathbf{A} \to \mathbf{B}$ then

$$\mathcal{F}_{\text{List}}(\mathbf{A}, \mathbf{B}, \phi_1) : \mathbf{List}(\mathbf{A}) \to \mathbf{List}(\mathbf{B})$$

Since Axiom's type constructors are functors, then category theory states this more simply as

$$\mathcal{F}_{\text{List}}(\mathbf{A}, \mathbf{B}, \phi_1) = \mathbf{List}(\phi_1)$$

Contravariance is a rarer case. However, Axiom's **Mapping** functor is contravariant in its first argument and covariant in its second. **Mapping** takes two types **A** and **B** and returns the type of all mappings from **A** to **B**.

As a concrete example for **Mapping**, suppose we wish to find the uniquely defined coercion.

$$\begin{gathered}\mathbf{Mapping}(\mathbf{Fraction}(\mathbf{Integer}), \mathbf{Fraction}(\mathbf{Integer})) \to \\ \mathbf{Mapping}(\mathbf{Integer}, \mathbf{Fraction}(\mathbf{Integer}))\end{gathered}$$

There exists a coercion

$$\iota : \mathbf{Integer} \to \mathbf{Fraction}(\mathbf{Integer})$$

the inclusion operator. There also exists a coercion

$$\text{id} : \mathbf{Fraction}(\mathbf{Integer}) \to \mathbf{Fraction}(\mathbf{Integer})$$

which is the identity operation. The uniquely defined coercion is as follows:

$$\begin{gathered}\mathcal{F}_{\text{Mapping}}(\mathbf{Fraction}(\mathbf{Integer})), \mathbf{Fraction}(\mathbf{Integer}), \mathbf{Integer}, \mathbf{Fraction}(\mathbf{Integer}), \iota, \text{id}) : \\ \mathbf{Mapping}(\mathbf{Fraction}(\mathbf{Integer}), \mathbf{Fraction}(\mathbf{Integer})) \to \\ \mathbf{Mapping}(\mathbf{Integer}, \mathbf{Fraction}(\mathbf{Integer}))\end{gathered}$$

which sends $f \mapsto \text{id} \circ f \circ \iota$.

The following definitions shows that there is a homomorphism image of a parameter in the created type. For example there is a homomorphic image of the underlying ring in any polynomial ring.

**Definition 10.6.9** *Let $f : (\sigma_1, \ldots, \sigma_n) \to \sigma$ be an n-ary type constructor.*
*If $(\forall i \in \{1, \ldots, n\})$(for some ground types $t_i : \sigma$) such that there exists a coercion*

$$\Phi^i_{f,t_1,\ldots,t_n} : t_i \to f(t_1, \ldots, t_n)$$

*then we say that $f$ has a* direct embedding *at its i-th position.*

*Moreover, let*

$$\mathcal{D}_f = \{i \mid f \text{ has a direct embedding at its } i-th \text{ position}\}$$

*be the* set of direct embedding positions *of $f$*

The definition of $\mathcal{D}_f$ is a technical definition of Weber's, needed for one of his assumptions (10.6.15).

### 10.6.10 Weber's work II: Assumptions and a conjecture

This section provides all the assumptions and results which Weber used in his "proof" of Weber's coherence conjecture 10.6.17 which will will state and at the end of this section. The assumptions and results are also require in our proof of the coherence theorem 10.6.25.

**Assumption 10.6.11** *For any ground type $t$, the identity on $t$ will be a coercion. The (well-defined) composition of two coercions is also a coercion*

This is clearly a sensible (if not-often implemented) statement. Since our coercions are always arrows of a category the above assumption must hold.

**Lemma 10.6.12** *If assumption 10.6.11 holds, then the set of ground types as objects together with their coercions as arrows form a category.*

**Proof**. Immediate.

The following assumption will provide us with the basis for a coherent type system. Our coherence is built by ensuring confluence amongst different paths leading to the same coercion. If we do not have coherence at the base types then we will not have coherence amonst the general types.

**Assumption 10.6.13** *The subcategory of base types and coercions between base types forms a preorder, i.e. if $t_1$, $t_2$ are base types and $\phi, \psi : t_1 \to t_2$ are coercions, then $\phi = \psi$.*

The following condition states that $\mathcal{F}_f$ is a functor over the category of all $f(\cdot, \ldots, \cdot)$s.

**Assumption 10.6.14** *Let $f$ be an $n$-ary type constructor which induces a structural coercion and let $f(t_1, \ldots, t_n)$, $f(t'_1, \ldots, t'_n)$, $f(t''_1, \ldots, t''_n)$ be ground types. Assume that the following are coercions.*

$$\begin{gathered}\iota \in \mathcal{M}_f \Rightarrow \phi_i : t_i \to t'_i, \phi'_i : t'_i \to t''_i \\ \iota \in \mathcal{A}_f \Rightarrow \phi'_i : t''_i \to t'_i, \phi_i : t'_i \to t_i \\ \iota \notin \mathcal{A}_f \cup \mathcal{M}_f \Rightarrow t_i = t'_i = t''_i \text{ and } \phi_i = \phi'_i = \mathrm{id}_{t_i}\end{gathered}$$

*Then the following conditions are satisfied:*

1. $\mathcal{F}_f(t_1, \ldots, t_n, t_1, \ldots, t_n, \mathrm{id}_{t_1}, \ldots, \mathrm{id}_{t_n}) = \mathrm{id}_{f(t_1,\ldots,t_n)}$
2. $\mathcal{F}_f(t_1, \ldots, t_n, t''_1, \ldots, t''_n, \phi_1 \circ \phi'_1, \ldots, \phi_n \circ \phi'_n) =$
   $\mathcal{F}_f(t_1, \ldots, t_n, t'_1, \ldots, t'_n, \phi_1, \ldots, \phi_n) \circ \mathcal{F}_f(t'_1, \ldots, t'_n, t''_1, \ldots, t''_n, \phi'_1, \ldots, \phi'_n)$

This is a condition which stops direct embeddings "becoming confused". Firstly, Weber declares that any type constructor can only have one direct embedding. (We will show how to relax this condition in a later section (10.6.26).) Secondly, he states that direct embeddings, where they exist, are unique.

**Assumption 10.6.15** *Let $f$ be an $n$-ary type constructor. Then the following conditions hold:*

1. $|\mathcal{D}_f| = 1$
2. *Direct embedding coercions are unique, i.e. if $\Phi^i_{f,t_1,\ldots,t_n} : t_i \to f(t_1, \ldots, t_n)$ and $\Psi^i_{f,t_1,\ldots,t_n} : t_i \to f(t_1, \ldots, t_n)$ then*

$$\Phi^i_{f,t_1,\ldots,t_n} = \Psi^i_{f,t_1,\ldots,t_n}$$

The following assumption is highly technical and shows how direct embeddings interact with structural coercions. Basically, they commute.

**Assumption 10.6.16** *Let $f$ be an $n$-ary type constructor which induces a structural coercion and has a direct embedding at its $r$-th position. Assume that*
*$f : (\sigma_1, \ldots, \sigma_n) \to \sigma$ and $f : (\sigma'_1, \ldots, \sigma'_n) \to \sigma$, and*
*$(\forall i \in \{1, \ldots, n\})(\exists t_i : \sigma_i$ and $t'_i : \sigma'_i)$. If there are coercions $\psi_r : t_r \to t'_r$, if the coercions $\Phi^r_{f,t_1,\ldots,t_n}$ and $\Phi^r_{f,t'_1,\ldots,t'_n}$ are defined, and if $f$ is covariant in its $r$-th argument, then the following holds:*

$$\Phi^r_{f,t'_1,\ldots,t'_n} \circ \psi_r = \mathcal{F}_f(t_1, \ldots, t_n, t'_1, \ldots, t'_n, \psi_1, \ldots, \psi_n) \circ \Phi^r_{f,t_1,\ldots,t_n}$$

$$\begin{array}{ccc} t_r & \xrightarrow{\psi_r} & t'_r \\ \downarrow{\scriptstyle \Phi^r_{f,t_1,\ldots,t_n}} & & \downarrow{\scriptstyle \Phi^r_{f,t'_1,\ldots,t'_n}} \\ f(t_1, \ldots, t_n) & \xrightarrow{\mathcal{F}_f(t_1, \ldots, t_n, t'_1, \ldots, t'_n, \psi_1, \ldots, \psi_n)} & f(t'_1, \ldots, t'_n) \end{array}$$

*However, if $f$ is contravariant in the $r$-th argument then:*

$$\mathcal{F}_f(t_1, \ldots, t_n, t'_t, \ldots, t'_n, \psi_1, \ldots, \psi_n) \circ \Phi^r_{f,t'_1,\ldots,t'_n} \circ \psi_r = \Phi^r_{f,t)_1,\ldots,t*n}$$

*or equivalently, the following diagram commutes:*

$$\begin{array}{ccc} t_r & \xrightarrow{\psi_r} & t'_r \\ \downarrow{\scriptstyle \Phi^r_{f,t_1,\ldots,t_n}} & & \downarrow{\scriptstyle \Phi^r_{f,t'_1,\ldots,t'_n}} \\ f(t_1, \ldots, t_n) & \xleftarrow{\mathcal{F}_f(t_1, \ldots, t_n, t'_1, \ldots, t'_n, \psi_1, \ldots, \psi_n)} & f(t'_1, \ldots, t'_n) \end{array}$$

We are now in a position to state Weber's coherence conjecture and his "proof". This attempts to show that when the aforementioned assumptions hold true, then we have a coherent type system. We will give more assumptions and a proper proof in section 10.6.18.

**Conjecture 10.6.17 (Weber's coherence conjecture)** *Assume that all coercions between ground types are only built by one of the following mechanisms:*

1. *coercions between base types*
2. *coercions induced by structural coercions*
3. *direct embeddings in a type constructor*
4. *composition of coercions*
5. *identity funciton on ground types as coercions*

*If assumptions 6.3.1, 6.3.3, 6.3.4, 6.3.5, and 6.3.6 are satisfied, then the set of ground types as objects, and the coercions between them as arrows form a category which is a preorder.*

This is the "proof" of this conjecture given in [Webe93b] and [Webe95].

Weber's "Proof". By assumption 6.3.1 and lemma 6.3.2, the set of ground types as objects and the coercions between them form a category.

For any two ground types $t$ and $t'$ we will prove by induction on max(com($t$),com($t'$)) that if $\phi, \psi : t \to t'$ are coercions then $\phi = \psi$.

If max(com($t$),com($t'$)) $= 1$ then the claim follows by assumption 6.3.3. Now assume that the inductive hypothesis holds for $k$, and let max(com($t$),com($t'$))$= k + 1$. Assume w.l.o.g. that $t \trianglelefteq t'$ and that $\phi, \psi : t \to t'$ are coercions.

Now $t \trianglelefteq t' \Rightarrow \text{com}(t) \leq \text{com}(t')$. So we may assume that $t' = f(u_1, \ldots, u_n)$ for some $n$-ary type constructor $f$.

By assumption 6.3.4 and the induction hypothesis, we can assume that there are ground types $s_1, s_2$ and unique coercions $\psi_1 : t \to s_1$ and $\psi_2 : t \to s_2$ such that either

$$\phi = \mathcal{F}_f(\ldots, t, \ldots, s_1, \ldots, \psi_1, \ldots)$$

or

$$\phi = \psi_1 \circ \Psi^i_{f,\ldots,s_1,\ldots}$$

Similarly either,

$$\psi = \mathcal{F}_f(\ldots, t, \ldots, s_2, \ldots, \psi_2, \ldots)$$

or

$$\psi = \psi_2 \circ \Psi^j_{f,\ldots,s_2,\ldots}$$

If $\phi$ is of form 6.1 and $\psi$ is of the form 6.3 then $\phi = \psi$ by assumption 6.3.4 and the uniqueness of $\mathcal{F}_f$.

If $\phi$ is of form 6.2 and $\psi$ is of form 6.3 then $\phi = \psi$ by assumption 6.3.6.

Analogously if $\phi$ is of form 6.1 and $\psi$ is of form 6.4.

If $\phi$ is of form 6.2 and $\psi$ is of form 6.4 then assumption 6.3.5 implies that $i = j$ and $s_1 = s_2$. Because of the induction hypothesis we have $\psi_1 = \psi_2$ and hence $\phi = \psi$ again by assumption 6.3.5. ■

### 10.6.18 The coherence theorem

In the above proof, there seem to be some irregularities.

#### On coercibility and complexity

Weber states in the proof that

$$t \trianglelefteq t' \Rightarrow \text{com}(t) \leq \text{com}(t')$$

Weber does not prove this, and indeed it is not true.

Suppose that $f$ is an $n$-ary type constructor and that it is contravarient in its $i$-th position. If

$$\mathcal{F}_f(s_1, \ldots, s_n, t_1, \ldots, t_n, \phi_1, \ldots, \phi_n) : f(s_1, \ldots, s_n) \to f(t_1, \ldots, t_n)$$

and $\phi_i : t_i \to s_i$ with

$$\text{com}(s_i) > \max(\text{com}(s_1), \ldots, \text{com}(s_{i-1}), \text{com}(s_{i+1}), \ldots, \text{com}(s_n))$$

In other words, $\text{com}(s_i)$ is the *unique*, maximum member of the set

$$\{\text{com}(s_j) \mid j \in \{1, \ldots, n\}\}$$

Then if $\text{com}(t_i)<\text{com}(s_i)$ and for all $j$ in $\{1, \ldots, n\}\backslash\{i\}$ we have that $s_j = t_j$, we have a counterexample to Weber's claim, that

$$\text{com}(f(s_1, \ldots, s_n)) > \text{com}(f(t_1, \ldots, t_n))$$

Thus Weber's assertion in invalid.

**Structural coercions (in the "proof")**

In equation 6.2 (and similarly equation 6.4) $\phi$ is given as a function

$$f(\ldots, t, \ldots) \to f(\ldots, s_1, \ldots)$$

however, $\phi$ is supposed to be a function of $t$.

**Structural coercions (syntax)**

Weber calls the structural coercion function from $f(s_1, \ldots, s_n)$ to $f(t_1, \ldots, t_n)$

$$\mathcal{F}_f(s_1, \ldots, s_n, t_1, \ldots, t_n, \phi_1, \ldots, \phi_n)$$

where $\phi_i$ is from $s_i$ to $t_i$ or $t_i$ to $s_i$ depending on whether $f$ is covariant or contravariant in the $i$th argument, respectively.

However, this is merely the functorial action of $f$ on the maps $\phi_1, \ldots, \phi_n$ and could be represented more compactly as

$$f(\phi_1, \ldots, \phi_n)$$

The source and target of each $\phi_i$ and knowledge of the sets $\mathcal{A}_f$ and $\mathcal{M}_f$ uniquely determine the source and target of $f(\phi_1, \ldots, \phi_n)$. This also demonstrates the uniqueness of $f(\phi_1, \ldots, \phi_n)$ and guarantees that assumption 6.3.4 holds.

**Identity coercions**

Weber states in assumption 6.3.1 that the identity function is a coercion. However, he never proves this to be unique. Indeed, automorphisms are perfectly natural maps $t \to t$.

In a computer algebra system like Axiom, many automorphisms are not automorphisms of the smallest category to which a type belongs.

For example, the ring-automorphism $\mathbb{Z}[X, Y] \to \mathbb{Z}[Y, X]$ is not a **PolynomialCategory** homomorphism since, for instance, the **leadingMonomial** function is not preserved under the map.

For some categories, like the category of groups, this may not be so easy to implement.

We add the following sensible assumption.

**Assumption 10.6.19** *The only coercion from a type to itself is the identity function.*

We will of course allow any number of functions from a type to itself, including conversions. It is merely the number of coercions which we are restricting.

### Composition of coercions

It is possibly symptomatic of the previous errors that Weber has neglected to cover all the possible case of $\phi$ and $\psi$.

All our coercions are compositions of coercions (or just a single coercion) from the list

1. coercions between base types
2. coercions induced by structural coercions
3. direct embeddings in a type constructor
4. identiy function on ground types as coercions

Suppose that $\phi = \tau_1 \circ \phi'$ and that $\psi = \tau_2 \circ \psi'$ where $\tau_1$ and $\tau_2$ are single coercions, and $\phi'$ and $\psi'$ are also (compositions of) coercions. $\phi'$ may be the identity coercion, as may be $\psi'$.

For a proof by induction on com, we need to cover all the cases of $(\tau_1, \tau_2)$ pairs. Thus the list we need to consider is

1. $\tau_1$ is a coercion between base types. $\tau_2$ is a coercion between base types.
2. $\tau_1$ is a coercion between base types. $\tau_2$ is a structual coercion
3. $\tau_1$ is a coercion between base types. $\tau_2$ is a direct embedding.
4. $\tau_1$ is a coercion between base types. $\tau_2$ is an identity function.
5. $\tau_1$ is a structural coercion. $\tau_2$ is a coercion between base types.
6. $\tau_1$ is a structural coercion. $\tau_2$ is a structural coercion.
7. $\tau_1$ is a structural coercion. $\tau_2$ is a direct embedding.
8. $\tau_1$ is a structural coercion. $\tau_2$ is the identity function.
9. $\tau_1$ is a direct embedding. $\tau_2$ is a coercion between base types.
10. $\tau_1$ is a direct embedding. $\tau_2$ is a structural coercion.
11. $\tau_1$ is a direct embedding. $\tau_2$ is a direct embedding.
12. $\tau_1$ is a direct embedding. $\tau_2$ is an identity function.
13. $\tau_1$ is an identity function. $\tau_2$ is a coercion between base types.
14. $\tau_1$ is an identity function. $\tau_2$ is a structural coercion.
15. $\tau_1$ is an identity function. $\tau_2$ is a direct embedding.
16. $\tau_1$ is an identity function. $\tau_2$ is a identity function.

Firstly notice we are only interested in the pairs as unordered entities. Thus some of these cases are duplicates.

Indeed: case 5 is case 2; case 9 is case 3; case 10 is case 7; case 13 is case 4; case 14 is case 8; case 15 is case 12. So we may discard cases 5, 9, 10, 13, 14, and 15.

Now notice that a base type can not have a direct embedding, neither can it induce a structural coercion. If either $\tau_1$ or $\tau_2$ is coercion between base types then the target of $\phi$ and

$\psi$ is a base type. Hence the other $\tau_i$ can not be a direct embedding nor can it be a structural coercion.

All the cases of this form are 2 (equivalently 5) and 3 (equivalently 9). Thus we may ignore these too.

Our remaining cases are 1, 4, 6, 7, 8, 11, 12, and 16.

To really prove the coherence theorem we are going to require some more assumptions. Only one of them (6.4.5) is difficult to justify.

We will replace this first assumption which uses Weber's definition 6.2.8 of a direct embedding presently with our own definition 6.4.3. In our definition assumption 6.4.2 will always hold (trivially).

**Assumption 10.6.20** *If a type constructor $f$ has a direct embedding at its $i$-th position and $f(t_1,\ldots,t_n)$ exists then $\Phi^i_{f,t_1,\ldots,t_n} : t_i \to f(t_1,\ldots,t_n)$*

Weber defines an $f$ to have a direct embedding (definition 6.2.8) at a particular position if there exists some $n$-tuple of types $(t_1,\ldots,t_n)$ for which $t_i$ directly embeds in $f(t_1,\ldots,t_n)$.

This seems sensible if we consider a type constructor to be a function which returns the carrier of the principal sort of the algebra. We are saying that direct embedding of a type in a constructor is equivalent to saying that a sort $\preceq$ is the principal sort.

It is true that an Axiom type constructor returns the carrier of the principal sort of the algebra but Axiom is more specific than that. The type constructor furnished with full complement of arguments *is* the principal sort of the algebra.

In fact it would be better to replace definition 10.6.9 and assumption 10.6.20 with the following definition.

**Definition 10.6.21** *Let $f : (\sigma_1,\ldots,\sigma_n) \to \sigma$ be an $n$-ary type constructor. If $(\forall i \in \{1,\ldots,n\})(\forall$ ground types $t_i : \sigma_i)$ there exists a coercion*

$$\Phi^i_{f,t_1,\ldots,t_n} : t_i \to f(t_1,\ldots,t_n)$$

*then we say that $f$ has a* a direct embedding at its $i$-th position.

*Moreover, let*

$$\mathcal{D}_f = \{i \mid f \text{ has a direct embedding at its } i-th \text{ position}\}$$

*be the* set of direct embedding positions of $f$.

The next assumption is undesirable due to its difficulty to guarantee in any implementation. However, it is provable in the more common covariant case (and we will prove it in the proof of the coherence theorem 10.6.25).

We are assuming that structural coercions behave confluently. The assumption below (10.6.22) is stated "too strongly". We will state the assumption we really need (10.6.23 – which is more complicated) immediately after. Assumption 10.6.22 gives the idea of what we require, whereas assumption 10.6.23 gives us what is necessary. It is an assumption about type constructors.

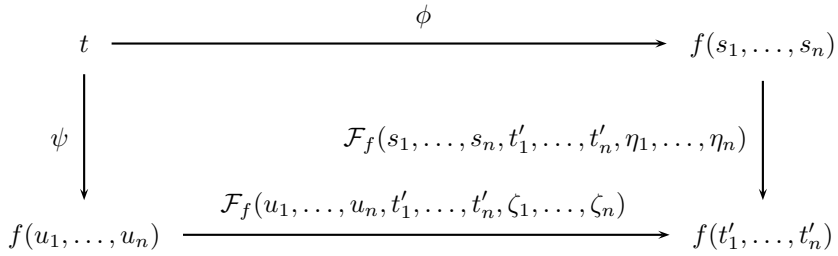
**Assumption 10.6.22** *Let the type constructor $f$ induce a structural coercion. If $\phi : t \to f(s_1,\ldots,s_n), \psi : t \to f(u_1,\ldots,u_n)$ are coercions and $t' = f(t'_1,\ldots,t'_n)$ then if there exists*

$$\mathcal{F}_f(s_1,\ldots,s_n,t'_1,\ldots,t'_n,\nu_1,\ldots,\nu_n) : f(s_1,\ldots,s_n) \to t'$$

$$\mathcal{F}_f(u_1,\ldots,u_n,t'_1,\ldots,t'_n,\eta_1,\ldots,\eta_n) : f(u_1,\ldots,u_n) \to t'$$

*the following holds*

$$\mathcal{F}_f(s_1,\ldots,s_n,t'_1,\ldots,t'_n,\eta_1,\ldots,\eta_n) \circ \phi = \mathcal{F}_f(u_1,\ldots,u_n,t'_1,\ldots,t'_n,\zeta_1,\ldots,\zeta_n) \circ \psi$$




Now, what we actually require. Suppose that a type $t$ is not constructed by the $n$-ary type constructor $f$ which is contravariant in its $i$-th position at which it also has a direct embedding.

Suppose also that $t$ is coercible to two types which may be directly embedding in $f$ (at the $i$-th position). Let us directly embed them to gain two new types (constructed by $f$). If these new types (constructed by $f$) can both be structurally coerced to the same type $t'$ (also constructed by $f$) then the two compositions of coercions $t \to t'$ are the same.

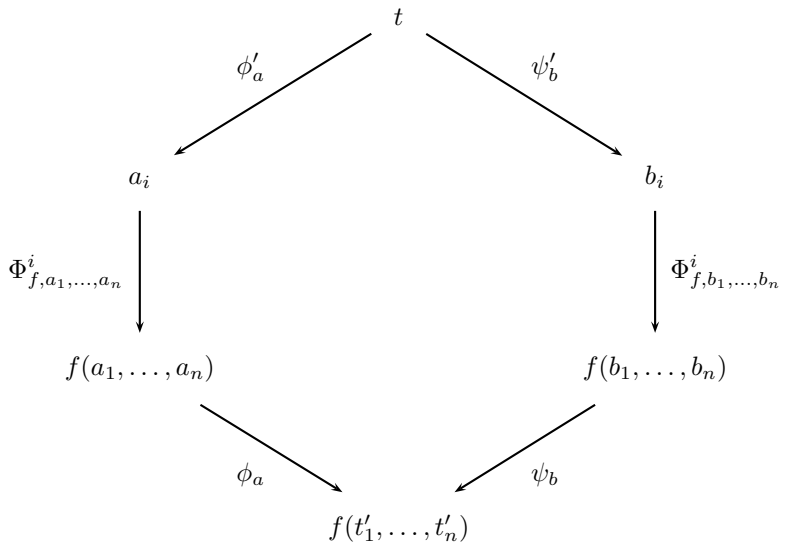
Succinctly, A structural coercion of a direct embedding of any other coercion is unique.

**Assumption 10.6.23** *Let $f$ be a type constructor contravariant at its $i$-th position at which it also has a direct embedding, and let $t$ be a type not contructed by $f$.*

*If $\phi'_a : t \to a_i$ and $\psi'_b : t \to b_i$ are coercions as $\phi_a : f(a_1,\ldots,a_n) \to f(t'_1,\ldots,t'_n)$ and $\psi_a : f(b_1,\ldots,b_n) \to f(t'_1,\ldots,t'_n)$ are structural coercions then*

$$\phi_a \circ \Phi^i_{f,a_1,\ldots,a_n} \circ \phi'_a = \psi_b \circ \psi^i_{f,b_1,\ldots,b_n} \circ \psi'_b$$

*or equivalently, the following diagram commutes*



Notice that assumption 10.6.23 is provable in the case that $f$ is covariant at $i$ (and we will prove this in the proof of the coherence theorem 10.6.25). In the contravariant case, there is no link between $t$ and $t'_i$ and it is this which make it an assumption.

The following assumption is more easily ensurable. We merely require that for two types constructed by the same type constructor, if there exists a coercion between them then it equals the structural coercions between them which we now assume to exist. This is also an assumption on type constructors.

**Assumption 10.6.24** *If $f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n)$ then there exists a structured coercion $f(s_1, \ldots, s_n)$ to $f(t_1, \ldots, t_n)$ and it is the unique coercion $f(s_1, \ldots, s_n)$ to $f(t_1, \ldots, t_n)$*

We are now finally in a position to state and prove the coercion theorem.

**Theorem 10.6.25 (Coherence theorem)** *Assume that all coercions between ground types are only built by one of the following mechanisms:*

1. *coercions between base types*
2. *coercions induced by structural coercions*
3. *direct embeddings (definition 10.6.21) in a type constructor*
4. *composition of coercions*
5. *identity function on ground types as coercions*

*If assumptions 10.6.11, 10.6.13,* **??**, *10.6.15, 10.6.16, 10.6.19, 10.6.23, 10.6.24 are satisfied, then the set of ground types as objects, and the coercions between them as arrows form a category which is a preorder.*

PROOF: By assumption 10.6.11 and lemma 10.6.12, the set of ground types as objects and coercions between them form a category.

All our coercions are either an individual coercion or a composition of finitely many coercions from the list: coercions between base types; structural coercions; direct embeddings (10.6.20) in a type constructor; and identity functions.

Thus we may decompose any coercion into such a finite composition. We define $\text{len}(\phi)$ (the *length* of $\phi$) to be the minimum number of coercions in any decomposition of $\phi$, Clearly, there are infinitely many values of $\text{len}(\phi)$ but there is a minimum value.

For any two ground types $t$ and $t'$ we will prove by induction on the length of the coercions between them that any coercions between them are unique.

Let $\phi, \psi : t \to t'$ be coercions.

If the length of $\phi$ and $\psi$ is 1 then (replacing the unordered pair $(\phi, \psi)$ with $(\tau_1, \tau_2)$) we need to look at our eight cases 1, 4, 6, 7, 8, 11, 12 and 16 defined above.

**Cases 1 and 4:** If $\phi$ is a base type coercion, then $t$ and $t'$ are base types and coercions between base types are unique by assumption 10.6.13.

**Case 8, 12, and 16:** If $\phi$ is an identity function on a ground type then $t = t'$ then by assumption 10.6.19 $\phi = \psi$.

**Case 6:** If $\phi$ and $\psi$ are structural coercions then $\phi$ and $\psi$ are equal by assumption 10.6.14.

**Case 7:** If $\phi$ is a direct embedding and $\psi$ is a structural coercion then there are two cases.

**Case 7a:** If $\psi$ is covariant. Then by assumption 10.6.24 $\psi = \phi$.

**Case 7b:** $\psi$ is contravariant. Thus $t = f(t_1, \ldots, t_n)$ and there exists $i$ and a coercion $t \to t_i$.

This can not happen since no composition of our four basic coercions can create such a coercion.

**Case11:** If $\phi$ and $\psi$ are direct embedings then by assumption 10.6.15 $\psi = \phi$.

We now may assume that any coercions of length less than or equal to $k$ are unique.

Suppose that $\phi, \psi : t \to t'$ are coercions and max(len($\phi$),len($\psi$))$= k+1$. Also suppose that $\phi = \tau_1 \circ \phi'$ and $\psi = \tau_2 \circ \psi'$ where for $i$ in {1,2} we have $\tau_i : s_i \to t'$ are single atomic coercions. Also $\phi' : t \to s_1$ and $\psi' : t \to s_2$ are unique coercions since their lengths are less than or equal to $k$.

If the length of $\phi$ or $\psi$ is 1 then we define $\phi'$ or $\psi'$ respectively to be the identity function on $t$.

As stated before, there are eight cases 1, 4, 6, 7, 8, 11, 12, and 16. (We may transpose $\tau_1$ and $\tau_2$ if we wish since their order is unimportant.)

**Case 1:** $\tau_1$ is a coercion between base types. $\tau_2$ is a coercion between base types. Then $t'$, $s_1$, and $s_2$ are all base types. Base types may only be the targets of base type coercions hence $t$ is a base type. Thus $\phi$ and $\psi$ are coercions between base types and are thus equal by assumption 10.6.13.

**Case 4:** $\tau_1$ is a coercion between base types. $\tau_2$ is an identity function. Again $t$ and $t'$ must be base types and $\phi$ and $\psi$ are equal by assumption 10.6.13.

**Case 6:** $\tau_1$ is a structural coercion. $\tau_2$ is a structural coercion. Let
$t' = f(t'_1, \ldots, t'_n)$

If $t$ is equal to some $f(t_1, \ldots, t_n)$ then $\phi$ and $\psi$ are structural coercions $f(t_1, \ldots, t_n)$ to $t' = f(t'_1, \ldots, t'_n)$ and hence are equal by assumption 10.6.14.

Else, we may assume that $t \neq f(t_1, \ldots, t_n)$. Since all our coercions are built from compositions of the four base types of coercion we may consider $\phi$ and $\psi$ to be chains of compositions. Without loss of generality, assume that the length of $\phi$ is $k+1$ and the length of $\psi$ is $l$ where $l \leq k+1$.

$$t = h_0 \stackrel{\phi_0}{\to} h_1 \cdots h_k \stackrel{\phi_k}{\to} h_{k+1} = t'$$

and

$$t = s_0 \stackrel{\psi_0}{\to} s_1 \cdots s_{l-1} \stackrel{\psi_{l-1}}{\to} s_l = t'$$

In both the composition chains of coercions from $t$ to $t'$ there must exist a minimal element of the chain which is constructed by $f$ and this (by assumption) can not be $s_0$ or $h_0$. Also because $\tau_1$ and $\tau_2$ are structural coercions, these minimal elements are not $h_{k+1}$ or $s_l$.

In the $\phi$-chain we will assume that this element is $h_a$. In the $\psi$-chain we will assume that this element is $s_b$.

Since $h_a$ is constructed by $f$ and $h_{a-1}$ is not and $\phi_{a-1}$ is one of the four basic coercions, it must be a direct embedding. (It can not be a structural coercion since $h_{a-1}$ is not constructed by $f$. It is not a coercion between base types because $h_a$ is constructed by $f$. It is not an identity coercion since $h_{a-1}$ is not constructed by $f$ whereas $h_a$ is.)

Similarly $\psi_{b-1}$ is also a direct embedding.

Now by assumption 10.6.15 the direct embedding must be at the same position. We will assume that this position is the $i$-th.

Let $h_a = f(a_1, \ldots, a_n)$ and $s_b = f(b_1, \ldots, b_n)$. Call the unique structural coercions
$\phi_a : h_a \to t'$ and $\psi_b : s_b \to t'$. (Notice that $h_{a-1} = a_i$ and $s_{b-1} = b_i$.)

Call the maps $\phi'_a : t \to h_{a-1}$ and $\psi'_b : t \to s_{b-1}$ where

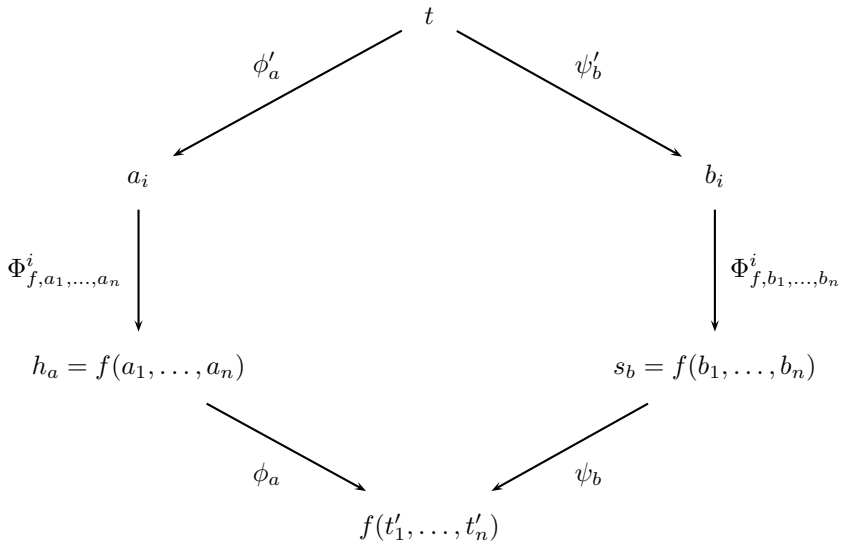
$$\phi'_a = \phi_0 \circ \cdots \circ \phi_{a-2}$$

and

$$\psi'_a = \psi_0 \circ \cdots \circ \psi_{b-2}$$

(If $a$ or $b$ equal 1 then we may insert an initial identity coercion and lengthen the chain by one. This does not affect our induction on length since we are always in good shape with identity coercions. Neither $a$ nor $b$ may be 0 by our assumption on $t$.)

Thus we have the following diagram, and we wish to prove that it commutes.




**Case 6a:** $f$ is covariant in the $i$-th position.

Consider the coercions $\alpha : a_i \to t'_i$ and $\beta : b_i \to t'_i$. (Recall that $t' = f(t'_1, \ldots, t'_n)$). These must exist because $\phi_a$ and $\psi_b$ lift them covariently respectively.

The definition of direct embeddings 10.6.21 guarantees the existence of $\Phi^i_{f,t'_1,\ldots,t'_n}$.

Now by the definitions above

$$\phi = \phi_a \circ \Phi^i_{f,a_1,\ldots,a_n} \circ \phi'_a$$

By assumption 10.6.16

$$\phi_a \circ \Phi^i_{f,a_1,\ldots,a_n} = \Phi^i_{f,t'_1,\ldots,t'_n} \circ \alpha$$

Thus

$$\phi = \Phi^i_{f,t'_1,\ldots,t'_n} \circ \alpha \circ \phi'_a$$

By induction on length we know that the coercion $t$ to $t'_i$ is unique. Hence

$$\alpha \circ \phi'_a = \beta \circ \psi'_b$$

So

$$\phi = \Phi^i_{f,t'_1,\ldots,t'_n} \circ \beta \circ \psi'_b$$

Assumption 10.6.16 gives us

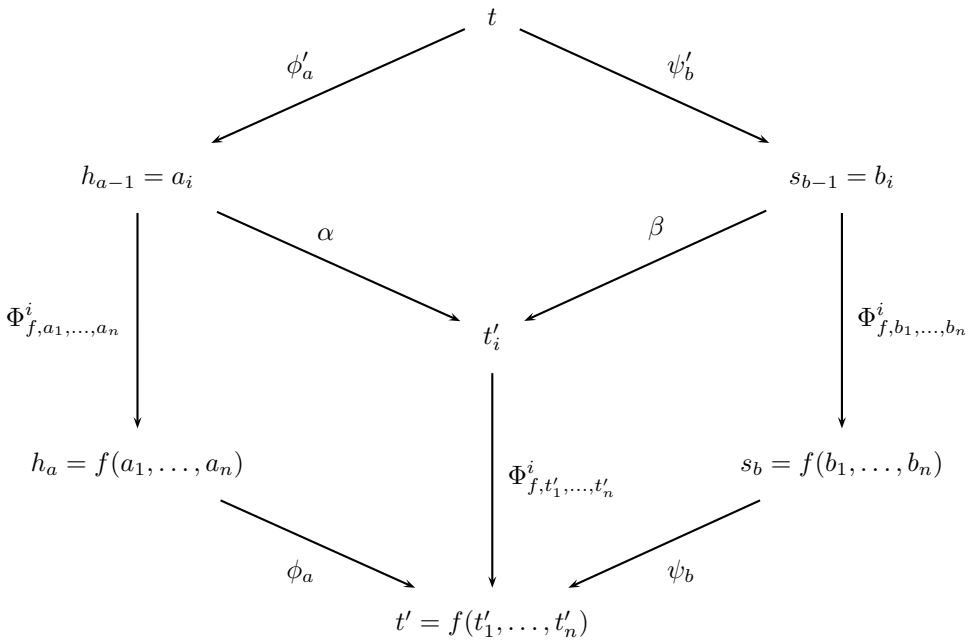
$$\Phi^i_{f,t'_1,\ldots,t'_n} \circ \beta = \psi_b \circ \Phi^i_{f,b_1,\ldots,b_n}$$

Thus

$$\phi = \psi_b \circ \Phi^i_{f,b_1,\ldots,b_n} \circ \psi'_b$$

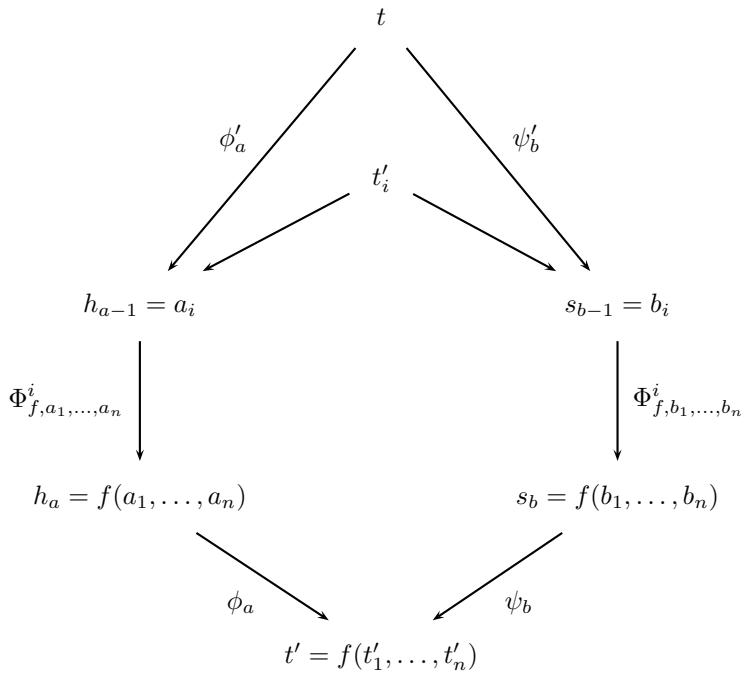
which by definition means $\phi = \psi$.

Graphically, the following is a commutative diagram.




**Case 6b:** $f$ is contravariant in its $i$-th position.

There is nothing to link $t$ and $'_i$ in this case and we must resort to assumption 10.6.23.




**Case 7:** $\tau_1$ is a direct embedding. $\tau_2$ is a structural coercion.

**Case 7a:** $\tau_2$ is covariant. Then $\phi = \psi$ by case 6a above.

**Case 7b:** $\tau_2$ is contravarient. Let $s_2 = f(s_{2_1},\ldots,s_{2_n})$ and $t' = f(t'_1,\ldots,t'_n)$. If $\kappa : t'_i \to s_{2_i}$, is the coercion lifted (contravariantly) by $\tau_2$ where

$$\tau_2 = \mathcal{F}_f(\ldots, s_2, \ldots, t'_i, \ldots, \kappa, \ldots)$$

Also

$$\tau_1 = \Phi^i_{f,t'_1,\ldots,\tau'_n}$$

The uniqueness of $\tau'$ implies that

$$\phi' = \Phi^i_{f,s_{2_1},\ldots,s_{2_n}} \circ \kappa \circ \phi'$$

By assumption 10.6.16 we have

$$\mathcal{F}_f(\ldots, s_2, \ldots, t'_i, \ldots, \kappa, \ldots) \circ \Phi^i_{f,s_{2_1},\ldots,s_{2_n}} \circ \kappa = \Phi^i_{f,t'_1,\ldots,s'_n}$$

Thus

$$\mathcal{F}_f(\ldots, s_2, \ldots, t'_i, \ldots, \kappa, \ldots) \circ \Phi^i_{f,s_{2_1},\ldots,s_{2_n}} \circ \kappa = \Phi^i_{f,t'_1,\ldots,s'_n} \circ \psi'$$

and therefore

$$\mathcal{F}_f(\ldots, s_2, \ldots, t'_i, \ldots, \kappa, \ldots) \circ \phi' = \Phi^i_{f,t'_1,\ldots,s'_n} \circ \psi'$$

Hence, $\phi = \psi$.

**Case 8:** $\tau_1$ is a structural coercion. $\tau_2$ is an identity function. Let $t' = f(t'_1,\ldots,t'_n)$. By viewing $\tau_2$ as the identity function

$$\mathcal{F}_f(t'_1,\ldots,t'_n,t'_1,\ldots,t'_n,\mathrm{id}_{t'_1},\ldots,\mathrm{id}_{t'_n})$$

Thus $\phi = \psi$ by case 6.

**Case 11:** $\tau_1$ is a direct embedding. $\tau_2$ is a direct embedding. Assumption 10.6.15 implies that the direct embeddings must be at the same position, $i$. Therefore $s_1 = s_2$. So by the inductive hypothesis $\phi' = \psi'$. By assumption 10.6.15 $\tau_1 = \tau_2$. Hence $\phi = \psi$.

**Case 12:** $\tau_1$ is a direct embedding. $\tau_2$ is an identity function. Let $t' = f(t'_1, \ldots, t'_n)$. By viewing $\tau_2$ as the identity structural coercion

$$\mathcal{F}_f(t'_1, \ldots, t'_n, t'_1, \ldots, t'_n, \mathrm{id}_{t'_1}, \ldots, \mathrm{id}_{t'_n})$$

Thus $\phi = \psi$ by case 6, above.

**Case 16:** $\tau_1$ is an identity function. $\tau_2$ is an identity function. Thus $\phi = \phi'$ and $\psi = \psi'$. However, since $s_1 = s_2$, $\phi' = \psi'$ by the induction hypothesis. Hence $\phi = \psi$.

Thus we have proved $\phi = \psi$ for all coercions of length less than or equal to $k+1$. Hence the result holds for all coercions, by induction. ■

In case 10b of the case when the length is 1 we claimed that there can not be built a coercion from a type constructor to one of its arguments.

From our description so far of Axiom, one might think that Axiom would permit the construction of the following type

**Fraction**(**Fraction**(**Integer**))

which is the quotient field of the quotient field of integers[17] However, the quotient field of a quotient field is itself and hence,

**Fraction**(**Fraction**(**Integer**)) ≅ **Fraction**(**Integer**)

For this reason, Axiom contains special code in the interpreter to stop such pathological types being instantiated. A function (**isValidType**) checks to see if one is trying to create a type like the one above, or

**Polynomial**(**Polynomial**(**Integer**))

But this is hand-crafted, special code covering a few cases and mentions the types by name. It is conceivable that a user could create a new type called **MyFraction** which is identical to **Fraction**. This would not be picked up by **isValidType** and thus

**MyFraction**(**MyFraction**(**Integer**))

could be instantiated. Since the type is then isomorphic to one of its arguments, it is feasible that a coercion between the two could be defined, contradicting our claim in case 10b of the case when the length is 1.

This coercion can still not be built from our four basic types, thus defining such a coercion contradicts our assumptions for the coherence theorem.

### 10.6.26 Extending the coherence theorem

First, Weber's "proof" of his conjecture 10.6.17 relies on induction on com($t$). This assumes that there are no types of infinite complexity in our system. This is not the case in Axiom, since one could define the following types (in one file):

[17] The quotient field of the integers ($\mathbb{Z}$) is more commonly known as the rationals ($\mathbb{Q}$).

**R(r : Ring) : Ring == r**
**D1 : Ring == R(D2)**
**D2 : Ring == R(D1)**

(though calling **1()$D1** would be disastrous[18])! So we add the following extra assumption.

**Assumption 10.6.27** *There do not exist any types of non-finite complexity*

The proof of the coherence theorem 10.6.25 does not rely on type having finite complexity. However it is still a sensible assumption to make and can be easily guaranteed in a real implementation. It is also imperative that this assumption holds if the automated coercion algorithm 10.7.11 is to terminate.

Assumption 10.6.15 states that we may have only one direct embedding into an algebra. However, in **Polynomial(Integer)** one would wish to perform the direct embeddings of both of **Integer** and **Symbol**, yet this violates assumption 10.6.15 and thus we would not be able to prove that our algebra system is coherent.

The reason for Weber's assumption is to stop coercions like the following occurring. If $A$ is a group, then being able to coerce $A \to A \times A$ whilst potentially useful, is ambiguous. As he points out in [Webe93b] and [Webe95]. $A$ can be coerced into $A \times A$ via the isomorphisms $A \cong A \times I$ or $A \cong I \times A$ (where $I$ is the trivial subgroup of $A$).

In this example the inclusions are ambiguous, since using either coercion, the target of the inclusion is a group. However, in many cases the types are "incomparable", (e.g. **Integer** and **Symbol**) and thus the assumption seems to be too strong. The question is what do we mean by "incomparable"?

Certainly, there is no coercion function **Integer** $\to$ **Symbol** or **Symbol** $\to$ **Integer**. But, for two distinct non-trivial, proper normal subgroups $G$, $H$ of $A$ such that $G \cap H = I$, then there is no coercion function $A/G \to A/H$, and coherency is lost. ($A/G$ is the quotient group "$A$ factored out by $G$" and is the set $\{aG \mid a \in A\}$ where $aG = a'G$ iff $a^{-1}a' \in G$). Thus, the condition of there not existing a coercion function between our two types is not sufficient for our definition of "incomparability".

However, we notice that there exists no type which can be coerced to both **Symbol** and **Integer**, but there does exist homomorphisms $A \to A/G$ and $A \to A/H$. So if we choose the statement "Types $A$ and $B$ are incomparable if there does not exist a type $S$ which can be coerced to both $A$ and $B$" as a definition of incomparability then we are in good shape.

We state this in the language of Weber as follows. To replace the assumption we first need to alter definition 10.6.21 (our previous replacement of definition 10.6.9 to the following.

**Definition 10.6.28** *Let $f : (\sigma_1, \ldots, \sigma_n) \to \sigma$ be an $n$-ary type constructor. If $(\forall i \in \{1, \ldots, n\})(\forall$ ground types $t_i : \sigma_i)$ there exists a coercion*

$$\Phi^i_{f,t_1,\ldots,t_n} : t_i \to f(t_1, \ldots, t_n)$$

*then we say that $f$ has a* direct embedding at its $i$-th position.

*Moreover, let $s$ be a ground type and $i \in \{1, \ldots, n\}$ and define,*

$$\mathcal{P}(i, s) := ((\exists \psi : s \to t_i) \wedge (\exists \Phi^i_{f,t_1,\ldots,t_n} : t_i \to f(t_1, \ldots, t_n)))$$

*where $\psi$ is a coercion, and also define,*

$$\overline{\mathcal{D}}_f := \{(i, s) \mid \mathcal{P}(i, s)\}$$

[18] This would try to create the constant 1 from the ring **D1** which can not be evaluated.

Note then that $\mathcal{D}_f = \{i \mid (\exists s)(\mathcal{P}(i,s))\}$.

We now alter assumption 10.6.15 to the following

**Assumption 10.6.29** *Let $f$ be an $n$-ary type constructor. Then the following conditions hold:*

1. $(i,s),(j,s) \in \overline{\mathcal{D}}_f \rightarrow i = j$
2. *Direct embedding coercions are unique. i.e. if $\Phi^i_{f,t_1,\ldots,t_n} : t_i \rightarrow f(t_1,\ldots,t_n)$ and $\Psi^i_{f,t_1,\ldots,t_n} : t_i \rightarrow f(t_1,\ldots,t_n)$ then*

$$\Phi^i_{f,t_1,\ldots,t_n} = \Psi^i_{f,t_1,\ldots,t_n}$$

So we may now extend the coherence theorem (theorem 10.6.25) by altering the assumption list to our new relaxed set of assumptions. It is not necessary to reprove the entire theorem, but merely the cases which involved assumption 10.6.15.

**Theorem 10.6.30 (Extended coherence theorem)** *Assume that all coercions between ground types are only built by one of the following mechanisms:*

1. *coercions between base types*
2. *coercions induced by structural coercions*
3. *direct embeddings (definition 10.6.28) in a type constructor*
4. *composition of coercions*
5. *identity function on ground types as coercions*

*If assumptions 10.6.11, 10.6.13, 10.6.14, 10.6.16, 10.6.27, 10.6.24, 10.6.19, 10.6.23, and 10.6.29 are satisfied, then the set of ground types as objects, and the coercions between them as arrows form a category which is a preorder.*

PROOF: The entire proof of theorem 10.6.25 is valid except where assumption 10.6.15 was used.

First we deal with the length of $\phi$ and $\psi$ being 1. The only case which relied on assumption 10.6.15 was case 11.

**Case 11:** If $\phi$ and $\psi$ are direct embeddings then by assumption 10.6.29 $\psi = \phi$.

Now, the induction case. The only cases which relied on assumption 10.6.15 were cases 6 and 11.

**Case 6:** In the proof of theorem 10.6.25 in the case we relied on assumption 10.6.15 to show that the two direct embeddings $h_{a-1} \rightarrow h_a$ and $s_{b-1} \rightarrow s_b$ were at the same position.

Now since $t \trianglelefteq h_{a-1}$ and $t \trianglelefteq s_{b-1}$ by assumption 10.6.29 the direct embeddings must be at the same position.

Thus the rest of the proof of this case holds.

**Case 11:** $\tau_1$ is a direct embedding. $\tau_2$ is a direct embedding. Since $t \trianglelefteq s_1$ and $t \trianglelefteq s_2$ assumption 10.6.29 implies that the direct embeddings must be at the same position, $i$. Therefore, $s_1 = s_2$. So by the inductive hypothesis $\phi' = \psi'$. By assumption 10.6.29 $\tau_1 = \tau_2$. Hence, $\phi = \psi$.

Inserting the rest of the proof of theorem 10.6.25 completes the proof. ■

Thus we have relaxed an important one of the conditions of the coherence theorem 10.6.25 and proved that the theorem still holds.

### 10.6.31 Conclusion

In this section we stated all the mathematics needed to state Weber's coherence conjecture and give his proof.

We have also stated extra mathematics to correct the statement of the conjecture and then prove it; hence promoting it to a theorem. We have then relaxed one of the conditions and shown that the theorem still holds.

This theorem is the cornerstone for ensuring correct coercions.

By having a coherent type system, then provided we are careful in how we implement our type constructors (they must satisfy the assumptions) and how we create our coercions (compositions of the four basic types) all our coercions are then unique.

## 10.7 The automated coercion algorithm

We will introduce some extra mathematics which will allow us to state the automated coercion algorithm. Furthermore, this sound foundation will allow us to demonstrate that all coercions generated by the algorithm are, in fact, homomorphisms. Indeed, we may even guarantee that these are coercions in the sense of definition 10.5.25.

### 10.7.1 Finitely generated algebras

In this section we will define what it means to say that an algebra is finitely generated, and also what it means to say that a finitely generated algebra is decomposable.

**Definition 10.7.2** *Let $\Sigma$ be an order-sorted $S$-signature. We say that $S$ has a* principal sort *if $\langle \Sigma, S \rangle$ defines an algebra which has one sort (which without loss of generality, we will always assume that to be $S_1$) which is the "most interesting" of the algebra.*

By "most interesting", we mean that a more naïve algebraist would consider this sort to define the entire algebra.

For example, if we define $\Gamma$ to be the Group algebra with sort,

$$(\text{"}\textit{the group}\text{"}, \text{"}\textit{a boolean sort}\text{"}, \text{"}\textit{an integer number system}\text{"})$$

then the "most interesting" sort is the one for "*the group*" which would be carried by a set of all the group elements.

All Axiom algebras must have a principal sort, referred to as %, in the source code. One can think of $S_1 = \%$ as the sort that the Axiom code "constructs". In the terminology of section 10.6 it is the type constructed by the type constructor. (Axiom views base types as type constructors with no arguments.)

**Definition 10.7.3** *We say that a theory $\langle \langle \Sigma, S \rangle, S \rangle$, is finitely generated iff $S$ has a principal sort, $S_1$, and the following set is finite,*

$$\ddot{\sum} := \bigcup_{n,q} \Sigma_{n,q,S_1}$$

This states that given a finitely generated Σ-algebra, $\langle A, \alpha \rangle$ (which is a model for $S$) then there are a finite number of operators which can return an element of the carrier of the principal sort.

This means that only finitely many functions (directly) construct (the carrier of) the principal sort.

The next definition allows us to decompose any element of such a Σ-algebra into at least two pieces. Moreover, there exists such a decomposition for each one of the (finitely many) constructors. Furthermore, decomposing such an element and recombining using the corresponding constructor is equivalent to the identity function.

**Definition 10.7.4** *A finitely generated theory $\langle \langle \Sigma, S \rangle, S \rangle$ is decomposable iff*

$$\begin{array}{l}(\forall \sigma \in \ddot{\Sigma})((\sigma \in \Sigma_{n,q,S_1}) \Rightarrow \\ \quad (\forall i \in \{1, \ldots, n\})(\exists \pi_{\sigma,i})(\pi_{\sigma,i} \in \Sigma_{1,S_1,q_i}) \wedge \\ \quad\quad ((\sigma(\pi_{\sigma,1}(\cdot), \ldots, \pi_{\sigma,n}(\cdot)) = \mathrm{id}_{S_1}) \in S))\end{array}$$

For example, in an algebra of lists, **cons** is a constructor. The corresponding functions which decompose an element are **car** and **cdr**. The following equation holds.

$$\mathbf{cons}(\mathbf{car}(\mathbf{x}), \mathbf{cdr}(\mathbf{x})) = \mathbf{x}$$

Notice that this equation is only well formed when **x** is not the empty list. For the empty list, which is constructed by the 0-ary function **nil()**, there are no decomposers, and the equation is as follows.

$$\mathbf{nil}() = \mathbf{x}$$

It is important to note that the part functions $\pi_{\sigma,i}$ are usually partial functions. For example, **car** is not defined on the empty list.

Notice that we have not yet defined a way of differentiating between which constructors construct which element. We will do this now.

### 10.7.5 Constructibility

We now come to the most important concept of this thesis. The aim of this thesis is to provde a method for creating coercions constructively. We now supply the means to do so.

As usual, we need to make some definitions first. We will state what we mean by a(n) (algorithmically) (re)constructible algebra. What we really mean is that an algebra is (algorithmically) (re)constructible if some subset of the elements of the most interesting sort-carrier of that algebra can be "built up" from a finite family of operators.

This finite family consists of "constructors" used to create increasingly "large" elements of the principal sort.

We will also have equations in the theory linking the constructors to "part functions" which we will use to split large elements into an $n$-tuple of smaller elements. The constructor applied to its associated part functions acting on an element will be equivalent to the identity function.

We will be able to tell how an element is constructed by using a "query function". That is to say we must be able to know which constructor(s) may construct any particular element of the principal sort.

We require these definitions so that if an algebra is a model of some algorithmically reconstructible theory, and another algebra contains some of the constructors of the first algebra, then some subset of the elements of the most interesting sort-carrier can be "coerced" from the first type to the second. We will show this to be a $\Xi$-homomorphism, where $\Xi$ is a particular signature to which both algebras belong.

We demand our extensions to be protecting extensions so that performing operations on an algebra (or looking at equations of elements of that algebra) when viewed as a model of the original theory or the extension of the theory yield "the same result".

**Definition 10.7.6** *We call a theory $\langle\langle\Sigma, S\rangle, S\rangle$ constructible iff $\Sigma$ is a protecting extension of a finitely generated theory. We then call $\Sigma$ a* constructing *signature, and any model for the theory, a* constructible algebra.

Similarly,

**Definition 10.7.7** *If $\langle\langle\Xi, X\rangle, \mathcal{X}\rangle$ is a decomposable theory extended by the reconstructible $\langle\langle\Sigma, S\rangle, S\rangle$, we define the* constructors *of $\langle\langle\Sigma, S\rangle, S\rangle$ to be*

$$\Sigma^C := \ddot{\Xi}$$

*We define the* constant constructors *to be the constructors of arity 0, denoted by $\Sigma^0$ and the* non-constant constructors *to be the constructors of all other arities. We denote this set by $\Sigma^{C-0} := \Sigma^C \backslash \Sigma^0$.*

*We also define the* part functions *of $\langle\langle\Sigma, S\rangle, S\rangle$ associated with $\sigma \in \Sigma^{C-0}$ to be the $\pi_{\sigma,i}$. (Constant constructors have no associated part functions.)*

*Finally we define the* constructor equations *to be the following set and demand it to be a subset of $S$ (and hence $\mathcal{X}$) given by*

$$\{(\sigma(\pi_{\sigma,1}(\cdot), \ldots, \pi_{\sigma,n}(\cdot) = \mathrm{id}_{S_1}) \mid \sigma \in \Sigma^{C-0}\}$$

We also need to ensure that certain other relationships between constructors which hold in our source algebra hold in our target algebra.

If an element of the source algebra may be constructed in more than one way, we require that reconstructing that element in the target algebra using any of those methods yields the same result.

Notice that since our theories are protecting extensions we are in good shape, since any equation that holds in the protecting extension which concerns only the sorts from the original (unextended) signature must hold in the original signature. Thus these equations will hold in the target algebra.

We call these equations that link different constructors together the secondary constructor equations.

**Definition 10.7.8** *We call a reconstructible theory $\langle\langle\Sigma, S\rangle, S\rangle$ algorithmically reconstructible iff : $S$ contains a boolean sort, $B$; $\Sigma$ contains a set of symbols, called the query functions, denoted and defined by*

$$\Sigma^Q := \{\mathcal{X}_\sigma \in \Sigma_{1,S_1,B} \mid \sigma \in \Sigma^C\}$$

*and $S$ contains the following equations, called the* query equations

$$(\forall \sigma_{n,q,S_1} \in \Sigma^C)(\mathcal{X}_{\sigma_{n,q,S_1}}(\sigma_{n,q,S_1}(a_1, \ldots, a_n)) = T)$$

*We also demand that for each* $\sigma_{n,q,S_1}$ *in* $\Sigma^C$

$$((\nexists\ a_1, \ldots, a_n)(\omega = \sigma_{n,q,S_1}(a_1, \ldots, a_n))) \Rightarrow (\mathcal{X}_{\sigma_{n,q,S_1}}(\omega) = F)$$

*Finally we demand that all secondary constructor equations are defined in the theory.*

*A* secondary constructor equation *is any equation which has a constructor as the final symbol on both right and left sides.*

We say "final symbol" to mean this is the operator applied last. In our notation (functions written on the left) a final symbol is written to the left of all the other elements of a formula.

For example, $\sigma()$ and $\sigma(e_1, \sigma'(e_2, e_3))$ both have $\sigma$ as a final symbol.

In the algorithm we will demand that the decomposable theory which both types model is the smallest theory which they both model. This is to ensure that the algorithm not only creates a homomorphism, but that it is a coercion (definition 10.5.25). The proof of this is given in corollary 10.7.21.

**Example 10.7.9** *We will take the variety of Lists as our example, and we will assume that* **List** *is a member of this variety. Then the constructors for the model are* **nil** *and* **cons**, *where* **nil** *is a constant constructor whereas* **cons** *is a non-constant constructor.* **cons***' associated part-functions are* **car** *and* **cdr**. *Thus we have the constructor equation*

$$\mathbf{cons}(\mathbf{car}(\mathbf{x}), \mathbf{cdr}(\mathbf{x})) = \mathbf{id}(\mathbf{x})$$

*The query functions are* **null** *and* **consp**, *and our query equations are*

$$\mathbf{null}(\mathbf{nil}) = T$$

*and for any list* $l$ *and any element of the underlying type,* $i$,

$$\mathbf{consp}(\mathbf{cons}(i, l)) = T$$

*Notice also that* $\mathbf{consp}(\mathbf{nil}) = F$ *and also, for the same* $l$ *and* $i$, $\mathbf{null}(\mathbf{cons}(i, l)) = F$

It is now worth discussing some of the finer points of homomorphism.

For $\phi$ to be a homomorphism we require that the constructor functions are preserved by homomorphism. We do not require that the part or query functions are homomorphically preserved; indeed we do not even require that they are in the signature of the algebra of the target of $\phi$.

Thus we do not require that both the source and target of $\phi$ are models of the same algorithmically reconstructible theory, but that the source is a model of an algorithmically reconstructible theory $\mathcal{T}$, and that all the constructors of $\mathcal{T}$ are functions of the target of $\phi$, inherited from the *same* signature (theory).

This is an important point. As an example, we will consider polynomial rings. A polynomial is a function such as

$$5x^2y^3 + 9xy^{45} - 34x + 7y^2 - 12$$

That is a sum of products of an element of the underlying ring (which in the above example might be the integers) and variables raised to non-negative powers. A monomial is a polynomial which is a product of a non-zero element of the underlying ring and variables raised to non-negative powers.

A polynomial ring is ordered by an extension of the order on the variables. In particular, in $\mathbb{Z}[x, y]$ if $x > y$ then a monomial $m_1$ is greater than another $m_2$ if the exponent of $x$ is greater in $m_1$ than in $m_2$. Should the exponent of $x$ be equal in both, then the exponent of $y$ is compared in the same manner.

The leading monomial of a polynomial is the largest monomial of a polynomial. If $x > y$ in the above polynomial then $5x^2y^3$ is the leading monomial. Should $y > x$ then the leading monomial would be $9xy^{45}$.

The reductum of a polynomial is the polynomial less its leading monomial.

Coercions between two polynomial rings need only use **leadingMonomial** as a part function (which is not preserved via homomorphism, since it depends on the ordering given to the variables) instead of some (unnatural) fixed (for all polynomial rings with variables from a fixed domain) "most-important monomial" function which would choose the same monomial, regardless of variable ordering.

Also, this allows us to form the natural homomorphism from $\mathbb{Q}[x]$ to $\mathbb{Z}(x)$, which in Axiom is

**Polynomial Fraction Integer** *to* **Fraction Polynomial Integer**

which are, in Axiom's view (without clever hackery in the interpreter) two unrelated **Rings**. This may be constructed *without* having to force + to be a constructor of **Fraction Polynomial Integer**, or indeed **leadingMonomial**, etc. to be available in **Fraction Polynomial Integer**.

### 10.7.10 The algorithm

We are now in a position to state the automated coercion algorithm.

The algorithm to create the coercion will be stated in English. It is too implementation dependent to state any finer.

The actual algorithm to coerce will be stated as Lisp pseudo-code. In Lisp, **(a b c)** means apply the function **a** to the arguments **(b,c)**. **cond** is like the switch statement in C or Java.

**cond** is equivalent to "if-then-else" statements – if a condition is true, then we evaluate and return the following statement (and leave the **cond** block), else go to the next condition and repeat.

For example, the line

$$((\alpha_{\chi_{\sigma_{0,\emptyset,S_1}}} x)\ (\beta_{\sigma_{0,\emptyset,S_1}}))$$

means

$$\textbf{if } \alpha_{\chi_{\sigma_{0,\emptyset,S_1}}}(x) \textbf{ then } \beta_{\sigma_{0,\emptyset,S_1}}()$$

The final statement **t**, is the default statement since **t** is always true. This line will only be reached when all the other conditions have not been satisfied, and shows that we have failed to build a total automated coercion function.

This could happen if one fails to list all the constructors (and their queries) for a type.

The lines of the form

$$((\alpha_{\chi_{\sigma_{0,\emptyset,S_1}}} x)\ (\beta_{\sigma_{0,\emptyset,S_1}}))$$

check for the constants of the type. For example, the constant polynomial 0 can not be constructed using a non-zero number of parts (using any normal construction methodology)

thus coercing 0 from $\mathbb{Q}[x]$ to $\mathbb{Z}(x)$ we might say

**if zero?(x)$Polynomial Fraction Integer**
**then zero()$Fraction Polynomial Integer**

returning the appropriate 0 in $\mathbb{Z}(x)$.

The lines of the form

$$((\alpha_{\chi_{\sigma_{n,q,S_1}}} x)(\beta_{\sigma_{n,q,S_1}}(\psi_{q_1}(\alpha_{\pi_{\sigma_{n,q,S_1},1}} x)\ldots(\psi_{q_n}(\alpha_{\pi_{\sigma_{n,q,S_1},n}} x))))$$

are the constructing lines. For example, in a polynomial ring we might write

**if x** is the sum of a monomial and a polynomial
**then coerce(leadingMonomial(x)) + coerce(reductum(x))**

where the addition function is that taken from the target domain.

In this case **leadingMonomial(x)** and **reductum(x)** are both polynomials. In general, the parts of the element need not be of the same type as the original element.

For example, in List algebras, coercing from **List(A)** to **List(B)** where there exists (or we can build) a coercion from **A** to **B** then we have (back in Lisp terminology)

((**consp x**) (**cons** (**coerce** (**car x**)) (**coerce** (**cdr x**))))

**(car x)** is the first element of the list and is of type **A** rather than **List(A)** – the type of **x**. So **(coerce (car x))** is of type **B** and thus may be **consed** on to the front of **(coerce (cdr x))** which is of type **List(B)**.

**Algorithm 10.7.11 (The Automated Coercion Algorithm)** *Let $\langle A, \alpha\rangle$ be a model for the algorithmically reconstructible theory $\langle\langle\Sigma, S\rangle, S\rangle$*

*Let $\langle B, \beta\rangle$ be a model of $\langle\langle\Delta, \mathcal{D}\rangle, \mathcal{D}\rangle$ where $\langle\langle\Sigma, S\rangle, S\rangle$ is a protecting extension of $\langle\langle\Delta, \mathcal{D}\rangle, \mathcal{D}\rangle$ and some (or all) of the constructors of $\langle\langle\Sigma, S\rangle, S\rangle$ are in fact $\langle\langle\Delta, \mathcal{D}\rangle, \mathcal{D}\rangle$.*

*Also we demand that there does not exist an extension $\langle\langle\Delta, \mathcal{D}\rangle, \mathcal{D}\rangle$ which both $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ model.*

*Then the following is an algorithm to coerce from the $\langle A, \alpha\rangle$ to the $\langle B, \beta\rangle$ model.*

*The $\psi_i$ for $i \neq 1$ are the (potentially automated) coercions from $A_i$ to $B_i$ from the abstract type of $S_i$.*

*The entire morphism created thus is called $\psi$ and not only is it a homomorphism, it is a coercion (definition 10.5.25).*

$\psi_1(x) :=$ (*cond*

$((\alpha_{\chi_{\sigma_{0,\emptyset,S_1}}} x)\ (\beta_{\sigma_{0,\emptyset,S_1}}))$

$\vdots$ *Repeat for each* $\sigma_{0,\emptyset,S_1}$ *in* $\Sigma^0$

$((\alpha_{\chi_{\sigma_{n,q,S_1}}} x)\ (\beta_{\sigma_{n,q,S_1}}(\psi_{q_1}(\alpha_{\pi_{\sigma_{n,q,S_1},1}} x))\ldots(\psi_{q_n}(\alpha_{\pi_{\sigma_{n,q,S_1},n}} x))))$

$\vdots$ *Repeat for each* $\sigma_{n,q,S_1}$ *in* $\Sigma^{C-0}$

$(t\ (error)))$

*The algorithm to create the coercion $\psi$ is*

$createCoerce(\langle A, \alpha\rangle, \langle B, \beta\rangle) :=$

$determine\ \langle\langle \Delta, \mathcal{D}\rangle, \mathcal{D}\rangle\ (error\ if\ doesn't\ exist)$

$determine\ \langle\langle \Sigma, \mathcal{S}\rangle, \mathcal{S}\rangle\ (error\ if\ doesn't\ exist)$

$determine\ \Sigma^0$

$for\ \sigma \in \Sigma^0$

$determine\ \alpha_{\chi_\sigma}$

$determine\ \beta_\sigma$

$determine\ \Sigma^{C-0}$

$for\ \sigma \in \Sigma^{C-0}$

$determine\ \alpha_{\chi_\sigma}$

$determine\ \beta_\sigma$

$determine\ \alpha_{\pi_{\sigma,i}}\ (for\ all\ relevant\ i)$

$construct\ and\ return\ \psi_1\ as\ defined\ above$

so the algorithm presented above allows us to algorithmically reconstruct elements of one type as elements of another.

### 10.7.12 Existence of the coercion

Before we prove that the automated coercion algorithm constructs a coercion when one exists we must prove that it does not create some unnatural function when no coercion exists.

We will use the terminology of algorithm 10.7.11 in this section.

In the simplest case, some of the functions will not exist in the target type and the algorithm will error at an early stage. This will be because there is no $\langle\langle \Delta, \mathcal{D}\rangle, \mathcal{D}\rangle$ which both $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ model.

If a homomorphism exists but not a coercion then this means that there exists an extension of $\langle\langle \Delta, \mathcal{D}\rangle, \mathcal{D}\rangle$ which both $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ model. In this case the algorithm will error at an early stage.

If all the constructors of $\langle A, \alpha\rangle$ are available in $\langle B, \beta\rangle$ but no homomorphism (coercion) exists then this could be because of (at least one of) the following causes.

#### One of the coercions to be used does not exist

At least one of the types which is required for construction or recursively required for construction (and which is not the carrier of the principal sort) may not be coercible to its counterpart in the target. Provided the algorithm checks at construction time that every coercion used directly or indirectly by $\psi_i$ exists (or is constructible) then we may report an error at an early stage.

**Non-homomorphic constructors**

The fact that $\langle\langle\Sigma, S\rangle, S\rangle$ is a protecting extension of $\langle\langle\Delta, \mathcal{D}\rangle, \mathcal{D}\rangle$ and that all secondary constructor equations hold in $\langle\langle\Sigma, \mathcal{S}\rangle, \mathcal{S}\rangle$ guarantees that these equations hold in $\langle\langle\Delta, \mathcal{D}\rangle, \mathcal{D}\rangle$ and hence in $\langle B, \beta\rangle$.

This not only ensures that the order of the lines in $\phi_1$ is unimportant but also that if a certain relationship holds in $\langle A, \alpha\rangle$ it *must* hold in $\langle B, \beta\rangle$.

For example, if someone defines that all finite field algebras are constructed by **0** and **succ** (the successor function) the automated coercion algorithm will not attempt to create the "coercion" $\mathbb{Z}_5 \to \mathbb{Z}_3$. This is because there exists a secondary constructor equation in the theory of finite fields of size $n$.

$$\mathbf{succ^n(0) = 0}$$

So in this example $\langle A, \alpha\rangle = \mathbb{Z}_5$ and $\langle B, \beta\rangle = \mathbb{Z}_3$. The equation $\text{succ}^5(0) = 0$ holds in the theory of finite fields of size 5. $\mathbb{Z}_5$ is obviously a model for this theory.

This equation is not true in any theory which the above theory extends and of which $\mathbb{Z}_3$ is a model. Otherwise the equation would hold in $\mathbb{Z}_3$ and that patently is not true.

Thus the automated coercion algorithm will error at any early stage from not being able to find $\langle\langle\Delta, \mathcal{D}\rangle, \mathcal{D}\rangle$.

### 10.7.13 Proving homomorphicity and coerciveness

We now make some notational definitions so that we may prove the final result of the section. That is, we will show that algorithm 10.7.11 constructs a homomorphism (theorem 10.7.20) which is a coercion (corollary 10.7.21).

Recall notation 10.4.11. Thus for any constructor symbol $\sigma_{n,q,S_1}$ the associated constructor function in $\langle A, \alpha\rangle$ is $\alpha_{\sigma_{n,q,S_1}}$ the query function is $\alpha_{\chi_{\sigma_{n,q,S_1}}}$ and for $i \in \{1, \ldots, n\}$ the part function $\alpha_{\pi_{\sigma_{n,q,S_1},i}}$

**Definition 10.7.14** *For a term $t$ in a term algebra $T_\Sigma(X)$ if*

1. $t \in X_s$ *then* $\text{length}(t) = 1$
2. $t \in \Sigma_{0,(),s}$ *then* $\text{length}(t) = 1$
3. $t = \sigma(t_1, \ldots, t_n)$ *then* $\text{length}(t) = 1 + \text{length}(t_1) + \cdots + \text{length}(t_n)$

The definition of length of an element will form the basis of our inductive proof of the automated coercion algorithm. However, in general, we will not be dealing with term algebras but their homomorphic images.

**Definition 10.7.15** *Let $x$ an element of a $\Sigma$-algebra, we define*

$$\text{length}(x) := \min\{t \in T_\Sigma(X) \mid \theta^*(t) = x\}$$

*where $\theta^*$ is the unique homomorphism given in the first universality theorem 10.4.23.*

**Assumption 10.7.16** *If one of the part functions $\alpha_{\pi_{\sigma_{n,q,S_1},i}}$ corresponds to $\alpha_{\sigma_1,(S_1),S_1}$ (thus $q_i = S_1$) then we demand that*

$$(\forall t \in T_\Xi(X)_1)(\text{length}(\alpha_{\pi_{\sigma_{n,q,S_1},i}}(\theta^*(t))) < \text{length}(\theta^*(t)))$$

*where $\theta^*$ is the unique homomorphism given in the first universality theorem 10.4.23*

A couple to technical definitions make the next assumption easier to understand.

**Definition 10.7.17** *Suppose $\langle A, \alpha\rangle$ is a constructed algebra, constructed by the $S$-sorted signature $\Sigma$. If for any $\imath \in \{2, \ldots, |S|\}$ we have that $S_i$ appears in the arity of any of the constructors of $\Sigma$, we say that $A_i$ is required for construction by $A_1$.*

So far example, in **List(Integer)** we have that **Integer** is required for construction since it is an argument of **cons**. Notice that **List(Integer)** is not required for consruction itself since it is the carrier of $S_1$.

**Definition 10.7.18** *Suppose $\langle A, \alpha\rangle$ is a constructed algebra, constructed by the $S$-sorted signautre $\Sigma$. Let $A_i$ be required for construction by $A_1$. Now consider $A_i$ as the carrier of the principal sort of $\langle B, \beta\rangle$, a constructed algebra.*

*If $B_j$ is required for construction by $B_1 = A_i$ (hence $j \neq 1$) then we say that $B_j$ is* recursively required for construction *by $A_1$.*

*Now suppose that the carrier $D_1$ of the principal sort of a constructed algebra $\langle D, \delta\rangle$ is recursively required for construction by $A_1$. If $D_k$ is required for construction by $D_1$, then we also say that $D_k$ is* recursively required for construction *by $A_1$.*

So for example in **List(List(Integer))**, the only type which is required for construction is **List(Integer)** since this is the only argument of **cons** which is not the carrier of the principal sort, **List(List(Integer))**.

Thus the only types which are recursively required for construction are **List(Integer)** and the types which are recursively required for construction by **List(Integer)**.

Since **Integer** is the only type required for construction by **List(Integer)** it is the only type recurively required for construction by **List(Integer)**.

Thus the types which are recurively required for construction by **List(List(Integer))** are **List(Integer)** and **Integer**.

The following assumption is required so that we may prove that the automated coercion algorithm terminates.

**Assumption 10.7.19** *We demand that $A_1$ is not recursively required for construction.*

**Theorem 10.7.20** *Let $\langle\langle\Sigma, S\rangle, S\rangle$ be an algorithmically reconstructible theory which is a protecting extension of the theory $\langle\langle\Xi, X\rangle, X\rangle$ such that $\Sigma^C \subseteq \ddot{\Xi}$.*

*Let $\langle B, \beta\rangle$ be a model for $\langle\langle\Xi, X\rangle, X\rangle$ and $\langle A, \alpha\rangle$ be a model for $\langle\langle\Sigma, S\rangle, S\rangle$.*

*Then the function $\psi$ given in algorithm 10.7.11 is a $\Xi$-homomorphism $\langle A, \alpha\rangle \to \langle B, \beta\rangle$*

PROOF:

Let $\phi$ be the correct homomorphism which $\psi$ is attempting to emulate.

Since $\psi_1$ contains finitely many cases, and covers all cases of construcctors for $\Sigma$, we need only consider one line from $\psi_1$. Also by induction on com$(t)$ we can assume that
$(\forall i \in \{2, \ldots, |S|\})(\psi_i = \phi_i)$

Since $\psi_1$ covers all cases of constructor for $\Sigma$ we do not need to consider the error line since it will never be reached.

For $\psi_1$ we may induct on length, and we are in good shape by assumption 10.7.16 (on length) and 10.7.19 (on the interaction between length and com).

Here is one line from $\psi_1$

$$(if\ (\alpha_{\sigma_q}\ a)\ (\beta_{\sigma_c}\ (\psi_{c_1}\ (\alpha_{p_1}\ a))\ldots(\psi_{c_n}\ (\alpha_{p_n}\ a))))$$

Now, assuming $(\alpha_{\sigma_q}\ a)$, we know that,

$$(\phi_1\ a) = (\phi_1\ (\alpha_{\sigma_c}\ (\alpha_{p_1}\ a)\ldots(\alpha_{p_n}\ a)))$$

then since $\phi$ is a $\Sigma$-homomorphism

$$(\phi_1\ (\alpha_{\sigma_c}\ (\alpha_{p_1}\ a)\ldots(\alpha_{p_n}\ a))) = (\beta_{\sigma_c}\ (\phi_{c_1}\ (\alpha_{p_1}\ a)\ldots(\phi_{c_n}\ (\alpha_{p_n}\ a)))$$

Now we know that, by our induction argument

$$(\beta_{\sigma_c}\ (\phi_{c_1}\ (\alpha_{p_1}\ a))\ldots(\phi_{c_n}\ (\alpha_{p_n}\ a))) = (\beta_{\sigma_c}\ (\psi_{c_1}\ (\alpha_{p_1}\ a))\ldots(\psi_{c_n}\ (\alpha_{p_n}\ a)))$$

whence for this line, and therefore every line and the entire function
$(\psi_1\ a) = (\phi\ a)$. So by induction, $\psi = \phi$ and thus $\psi$ is a homomorphism. ■

Now by adding one extra condition, we may prove that the automated coercion algorithm generates coercions in the sense of definition 10.5.25.

**Corollary 10.7.21** *Let $\langle\langle\Sigma, S\rangle, S\rangle$ be an algorithmically reconstructible theory which is a protecting extension of the theory $\langle\langle\Xi, X\rangle, X\rangle$ such that $\Sigma^C \subseteq \ddot{\Xi}$*

*Let $\langle B, \beta\rangle$ be a model for $\langle\langle\Xi, X\rangle, X\rangle$ and $\langle A, \alpha\rangle$ be a model for $\langle\langle\Sigma, S\rangle, S\rangle$ such that there does not exist any extension of $\langle\langle\Xi, X\rangle, X\rangle$ which both $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ model.*

*Then the function $\psi$ given in algorithm 10.7.11 is a coercion $\langle A, \alpha\rangle \to \langle B, \beta\rangle$*

PROOF: No other theories extend $\langle\langle\Xi, X\rangle, X\rangle$ which both $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ model so the only thing which could stop $\psi$ being a coercion would be for there to exist a theory which both $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ were to model which was not extended by $\langle\langle\Xi, X\rangle, X\rangle$.

So, suppose (for a contradiction) $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ were to model a theory $\Omega$ which $\langle\langle\Xi, X\rangle, X\rangle$ does not extend, then we may manufacture a new theory containing all the sorts, operator symbols, and equations of both theories.

We may have some duplication of sorts and operators, so this manufacturing process would need to be performed intelligently. Explicitly, $\Omega$ and $\langle\langle\Xi, X\rangle, X\rangle$ may both be extensions of some theory $\Theta$. We might (but not always[19]) only wish our manufactured theory to contain the sorts, operator symbols, and equations $\Theta$ once, not twice.

This new theory would clearly extend $\langle\langle\Xi, X\rangle, X\rangle$ and $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ would both model this new theory. Hence we have a contradiction and such an $\Omega$ can not exist.

So $\langle\langle\Xi, X\rangle, X\rangle$ must specify the unique "smallest" variety to which both $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ belong. We already know by theorem 10.7.20 that $\psi$ is a $\Xi$-homomorphism; by definition 10.5.25 it must be a coercion.

This is why algorithm 10.7.11 looks for $\langle\langle\Delta, \mathcal{D}\rangle, \mathcal{D}\rangle$, since it must specify the smallest variety to which both $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ belong.

[19] The theory of rings inherits from the theory of monoids twice

### 10.7.22 Conclusion

In conclusion, we have shown that it is possible to create a homomorphism between two types from an important subset of types.

Moreover, this homomorphism may be constructed algorithmically and hence the general construction of homomorphisms is implementable on a computer.

Furthermore, if the type system adheres to all the assumptions made, and the homomorphism is from the theory which specifies the smallest variety to which both types belong, then it is the unique *coercion* between the two types.

## 10.8 Implementation Details

The basic design was as follows. To perform a coercion from a **Domain A** to a **Domain B**, one needs to be able to ascertain which constants, queries, constructors, and part functions are available to both **A** and **B**. These functions should all come from the same algorithmically reconstructible theory (**Category**).

The automated coercion function should be created and then compiled and stored away ready for fast access in case of future use. Such items are called "cached lambdas" or "clams", for short, and are already part of the Axiom system. Thus this was implemented.

The automated coercion function should be integrated with the current coercion mechanism in Axiom. This was done by inserting the relevant line at the correct point of the function **coerceInteractive**.

### 10.8.1 Labelling operstors

We had to decide on a mechanism to mark which operators were constructors, part functions, etc.

One method would have been to create sub-types of the Axiom **Domain**, **Mapping**, which would have been infeasible, requiring much rewriting of Spad compilers. This is also true of the method given in a later section.

A Spad **Domain** contains the following parts:

1. the name of the type constructor
2. a (variable name, **Category**) pair[20] for each[21] of the parameters of the type constructor
3. a **Category** or **Join** (intersection) of **Categories** to which the **Domain** belongs
4. a list of additional operator symbols (functions) for the type
5. a list of methods for (some of or all) the operator symbols of the type

Every operator symbol definition in a **Domain** or **Category** (and hence available in a **Domain** of that **Category**) has a certain number of special comments called ++ comments.

[20] Occasionally a (variable name, **Domain**) pair. For example, **IntegerMod(p:PositiveInteger)**

[21] This is not the same as all the sorts of the signature. One may declare extra unused sorts. Also we may declare a function where the source of the target may contain a ground type. For example, functions which return a boolean-like type are usually declared to return **Boolean** rather than a pair where the **Category** is the **Category** of boolean-like types.

These ++ comments (as opposed to ordinary comments in most languages and Axiom's other style of comments, −− comments) are parsed by the compiler to produce the documentation for Axiom's sophisticated on-line help system, **HyperDoc**. (**HyperDoc** was one of the first hypertext systems.)

This parsing of the ++ comments allowed us to enter special keywords for each special function which were read and stored at compile time for each **Domain**. We could then read them at run-time using the Axiom interpreter's built in ++ comments reader.

### 10.8.2 Getting information from domains

The ++ comments were gathered for a **Domain** by recursion up the **Category** inheritance lattice, using the **GETDATABASE** function. We asked the database for all the documentation for each **Category**, and then asking which **Categories** it extended. The documentation includes all the functions and their respective ++ comments. This allowed the automated searching for keywords, and hence, the special functions of a **Domain**.

Note that we only needed the comments for functions (operator symbols) declared in a **Category**. Functions declared in **Domains** are of no interest to the automated coercion algorithm. This is because these do not correspond to any of the special functions (constructors, part functions, or queries) of a signature or theory.

The keywords were checked quite simply, being members of the following list:

$$\mathbf{list}('\text{"}\mathbf{constant}\text{"},'\text{"}\mathbf{constructor}\text{"},'\text{"}\mathbf{part}\text{"},'\text{"}\mathbf{query}\text{"})$$

Part number checking was only slightly more difficult with each number having to be converted from a (Lisp) string to a (Lisp) integer. A restriction was placed on queries that they must be called the same as their associated constant (or constructor), but with a "?" appended. This restriction was not necessary and could have been worked around easily.

The special function lists from both **Domain**s were compared and the required functions were extracted from both.

### 10.8.3 Checking information from domains

For each special function, a check was performed to see whether that function was really available in the **Domain**.

The envisaged problem was that Axiom's **Categories** can be conditional: that is to say, some function definitions only exist if certain relationships hold for **Category**'s parameters.

**GETDATABASE** ignores the parameters to a **Category** and returns all the functions which may be exported by a **Domain** of that **Category**. Hence, the extra check was necessary to ensure that the automated coercion function did not try to include these unavailable functions.

Thus the homomorphism created by the automated coercion algorithm was always equivalent to definition of a conditional homomorphism given in section 10.5.10.

### 10.8.4 Flaws in the implementation

The implementation was originally envisaged as a fully working piece of code of production standard and hence shippable with a commercial release of Axiom. For various reasons

outlined below, only a working prototype was implemented.

Deliberate restrictions placed by the author on the deisgn were:

1. Query function names were restricted to be the name of their associated constant function with a "?" appended.
2. The number of (non 0-ary) constructors was limited to one
3. Recursion through **coerceInteractive** was achieved using a naïve means
4. Neither the existence of $\langle\langle\Sigma, S\rangle, S\rangle$ nor $\langle\langle\Delta, \mathcal{D}\rangle, \mathcal{D}\rangle$ were checked. (See algorithm 10.7.11 for the meaning of these).

Both the first two items could have easily been worked around but would have required a more complex ++ comment reader. This was considered to be a minor detail, which would have required too much implementation time.

The third item was more problematic. Attempts were made to remedy this situation. However, the depth of knowledge needed to implement this correctly would have required too much time to learn. This was the main factor in the downgrading of the implementation to mere prototype status.

The method to work around the complexities of Axiom's evaluation loop was to add a (Spad) **Package** which exported a function which then called the Lisp function. The code for this **Package** (called **NCoerce**) follows.

```
--% Coercion Package
)abbrev package NCOERCE NCoerce
NCoerce(Source:Type,Target:Type):Exports == Implementation where
  Exports ==> with
    nCoerce: Source -> Target
  Implementation ==> add
    nCoerce(x:Source):Target ==
      nCreateCoerce(x,Source,Target)$Lisp
```

To call a function from a **Package** is relatively easy.

The function (**nCoerce** above) then uses the Axiom **$Lisp** syntax for calling a function.

One item is a bit more difficult to solve. The existence of $\langle\langle\Sigma, S\rangle, S\rangle$ is implicit if a **Domain** has special functions. It would also be possible to check if both source and target are members of a common **Category** (using **GETDATABASE**). It would even be possible to check that all the constructors of the source are from this **Category**. Finally, it would be possible to ensure that this is the smallest such **Category**.

However, it would be most problematical to determine whether $\langle\langle\Sigma, S\rangle, S\rangle$ is a protecting extension of $\langle\langle\Delta, \mathcal{D}\rangle, \mathcal{D}\rangle$ or not.

This is a more general problem than one which just applies to the automated coercion algorithm. This is due to the fact that Axiom believes everything that you say. One may produce a **Category** and claim that it extends another. Yet if Axiom's **Categories** were really varieties, specified by theories there would need to be some way of checking whether the equations in the original **Category** still held in the new one. Similarly, **Domains** need not model the theory specifing the variety (**Category**) to which they have been declared to belong.

The current situation in automated theorem proving means that Axiom will believe any false assertions like those mentioned in the previous paragraph. This is not a design flaw of either Axiom or the automated coercion algorithm; neither Axiom's designers nor I had any choice – automated theorem proving has not yet advanced sufficiently for us to take advantage of it. Similarly (and moreover) Axiom will believe that any function (which one may implement) called **coerce** is a valid coercion.

This means that in this implementation of the automated coercion algorithm in the current release of Axiom, it would be possible to create new "coercions" between (potentially) unrelated types or non-homomorphically between similar types. Using the example from section 10.7.12 we could create a "coercion" $\mathbb{Z}_5 \to \mathbb{Z}_3$

The present state of automated theorem proving technology with special reference to Axiom is detailed in [Mart97]. This paper details recent research and near-future directions for the subject.

Other flaws which will be dealt with next including:

1. Some Axiom **Categories** do not agree with our notion of varieties (specified by signatures or theories) (Sections 10.9.1, 10.9.3, 10.9.4)
2. Axiom **Domains** have too much in common with their **Categories**. There is a lack of distinction between operator symbols and operators. Similarly between sorts and carriers (Sections 10.9.2, 10.9.5)

### 10.8.5 Conclusion

We conclude that the automated coercion algorithm is probably implementable in the Axiom interpreter. It is certainly possible to create a version which works. However, it would take someone with deeper knowledge of the code to force a complete integration of the current implementation with the Axiom interpretation.

## 10.9 Making Axiom algebraically correct

We will discuss various details which need to be changed to make a computer algebra system more "algebraically correct". By this we mean that Axiom's **Categories** behave more like our concept of varieties (specified by order-sorted theories) and that the **Domains** act more like our notion of order-sorted algebras more correctly.

### 10.9.1 Explicitly defined theories

In Axiom, the **Categories** were originally intended to be akin to varieties specified by signatures. The **Domains** were then meant to be like algebras of those signatures.

The **Categories** do behave and look to the casual observer like they were specified by multi-sorted algebraic type theories, but there are some clear differences.

### 10.9.2 Operator symbols and names

In Axiom, if a category defines an operator symbol $\sigma_{n,q,s}$ then for all domains in that category, that operator name $\alpha_{n,q,s}$ of $\sigma_{n,q,s}$ will be $\sigma_{n,q,s}$.

It may appear to be not that much of a disadvantage to have such a restriction on operator names, but an important case in mind is that of the monoid. A ring, has two binary operators over which it forms a monoid. The only way in Axiom that we can ensure this is to have two different monoidal categories. A distinct disadvantage.

If we allowed operator symbols to differ from operator names then another **Category** could extend **Monoid** in two different ways.

Syntactically this would be most difficult. We would need a way of declaring the two operator symbols in **Ring** (traditionally + and $*$) as being from different operator symbols in **Group** (which extends **Monoid**) and **Monoid**.

For example we could write something like the following

**Ring:Category == with (M:Monoid, G:Group)**

**−− Operators from Monoid**
**∗: (∗:(%,%) − > %)$M**
**1: (1:() − > %)$M**

**−− Operators from Group**
**+: (∗:(%,%) − > %)$G**
**0: (1:() − > %)$G**
**inv: (inv:% − > %)$G**

...

This would identify $*$ and 1 with the binary operator and identity element, respectively, from **Monoid**. Whereas +, 0, and **inv** would be identified with the binary operator, identity element, and inverse function from **Group**.

Notice that both the operator symbol (e.g. $*$) and both the source arity (e.g. (%,%)) and target (e.g. %) sort are needed to uniquely identify the operator. These correspond to $\sigma$, $q$ and $s$ respectively. (we do not need to know $n$ since this is deducible from 1).

A function in Axiom (in either a **Domain** or **Category**) is currently represented by (**name** (**target source**) **comments**). Function declarations in **Categories** are normally of the form[22]

$$(\sigma\ (s\ q)\ \textbf{comments})$$

and once exported by a ground type (i.e. genuine Axiom **Domain** with all parameters to the **Domain** fixed) it becomes

$$(\alpha_{n,q,s,\sigma}\ (A^s\ A^q)\ \textbf{comments})$$

where $\alpha_{n,q,s,\sigma} = \sigma$.

For the coercion algorithm, we inserted special words in the comments field to try and mimic an operator symbol.

Another approach might have been to declare other types e.g. **Constructor(a,b)** which are sub-types of **Mapping(a,b)**.

A far better approach is to extend the information contained within the function construct. For initial definitions in **Categories** – that is new operators which have come from this

[22] $s$ and $q$ are reversed in Axiom's internal representation of a function.

**Category**, not ones which this **Category** may extend – then the current declaration method suffices.

However, as in the above **Ring** example for **Categories** which extend others then a better method may be

$$(\sigma' \ (\sigma \ (s \ q) \ \Sigma) \ \textbf{comments}$$

where $\sigma'$ is the new operator symbol corresponding to $\sigma$ a member of $\Sigma_{n,q,s}$ where this signature extends $\Sigma$.

We have used variable names to avoid confusion. One may wish to define a **Category** called **DoubleMonoid** where **G** is a **Monoid** instead of a **Group**. (Obviously no **inv** operator would be available to such a **Category**.)

The same terminology could also be used to define specific algebras (**Domains**). For example, for all $n$ in $\mathbb{N}$, $n\mathbb{Z}$ (whose elements is the set $\{nz \mid z \in \mathbb{Z}\}$) is an additive group, whereas $S(n)$ (the symmetric group on $n$ elements) is usually considered to be a multiplicative group.

So, for $n\mathbb{Z}$ we might write

**IntegersTimes(n:PositiveInteger):Group** ==

**+ : (* : (%,%) − > %)**
**0 : (1 : () − > %)**
**- : (inv : % − > %)**
...

However for $S(n)$ we might write:

**Symmetricgroup(n:PositiveInteger):Group** ==

*** : (* : (%,%) − > %)**
**() : (1 : () − > %)**
**inv : (inv : % − > %)**
...

In this example we can see the following proposed methodology for declaring operator names which correspond to operator symbols.

$$(\alpha_{n,q,s,\sigma} \ (\sigma \ (s \ q)) \ \textbf{comments})$$

The signature $\Sigma$ does not need to be mentioned in each function definition since the algebra is only declared to be a model of one signature.

This does create more "unnecessary" confusion for **Domain/Category** writers, but correctness should overrule ease-of-use. It is also conceivable that this will have a negative effect on the amount of time it takes to compile a **Domain**. However, this is true of all type-checking compilers.

More inportantly, it should not have any effect on run-time speed of execution. This should definitely be the case if the internal representation is moved to something approximating Axiom's current order.

$$(\alpha_{n,q,s,\sigma} \ (s \ q) \ \sigma \ \textbf{comments})$$

This puts the target and source closer (and less operations away) from being discovered.

Clearly, in this proposed methodology the carriers of the same source and target are not mentioned explicitly in the function representation.

To speed up function-type look-ups (modemap selection) each **Domain** could provide a hash of sorts and carriers. Indeed, then the internal representation could be

$$(\alpha_{n,q,s,\sigma}\ (A^s\ A^q)\ \sigma\ \textbf{comments})$$

and then modemap selection would be as fast as in the current version of Axiom, but the hash could be used (going in the other direction) to determine $s$ and $q$ for the automated coercion algorithm.

### 10.9.3 Moving certain operators

Another factor which stops Axiom acting totally homomorphically, is that certain operators appear in **Categories** too far "up" the inheritance lattice. This is best illustrated by an example.

In Axiom, one often would like to be able to convert **Lists** to **Sets**. (The **Domain** whose items are finite sets (or actually, sometimes classes), over some particular type is called **Set** in Axiom.)

**Sets** and **Lists** in Axiom are both certainly finite collections which can be built by adding in another element at a time. In **List** this may be achieved using either **cons** or **append** (though obviously, **cons** is far more efficient) whereas **Sets** can be built using the (sometimes non-effective) command **insert**[23].

Axiom knows that **Lists** are sorted whereas **Sets** are not. The problem, which is immediately obvious, is that adding in a new element to a **List** will always increase the length of the **List**. This is not true of (Axiom's finite) **Sets**. They both however get the same element-counting-operator from the same **Category**. This function, # comes from the **Category**, **Aggregate**, which is the most general type of "collection" in Axiom.

**List(S)** (for some fixed **S**) is a model of **ListAggregate** and **Set(S)** is a model of **FiniteSetAggregate**. Both of these theories are extensions of the theory **Aggregate**.

Therefore if we had a homomorphism, $\phi$ from **Lists** to **Sets**, then the following equation should hold, yet it clearly does not,

$$\phi(\#([1,1])) = \#(\phi([1,1]))$$

since the left hand side is,

$$\phi(2) = 2$$

and the right hand side is,

$$\#(\{1\}) = 1$$

and unless the carrier sort of collection length is not **NonNegativeInteger** but a different type, one in which all elements are equal, we will not be able to create any homomorphisms from **List** to **Set**.

In section 10.7.5 we demanded that if the automated coercion algorithm was to be applicable (in this case) from **List(S)** to **Set(S)** then **ListAggregate** would need to be a protecting extension of **Aggregate**[24].

[23] This operation inserts a (potentially) new element into a set. $\textbf{insert}(x,s) = s \cup \{x\} = s; x$

[24] Or an extension of **Aggregate** which both **ListAggregate** and **FiniteSetAggregate** extends.

It is clear that in **ListAggregate** the following equation holds

$$\#(\mathbf{cons}(a, b)) = 1 + \#(b)$$

If this were a protecting extension then this equation would hold in **Aggregate**. However, as we have already observed.

$$\#(\mathbf{insert}(a, b) \neq 1 + \#(b)$$

when $a \in b$.

Thus **ListAggregate** is not a protecting extension of **Aggregate** and the automated coercion algorithm can not be applied.

There are three obvious solutions:

1. Disallow coercions from **List** to **Set**. Although we would still allow a non-homomorphic **convert** operator
2. Move the element-counting-operation further down the **Category** inheritance lattice until it does not appear in any **Categories** to which *both* **List** and **Set** belong. Or if they do both belong to this **Category**, ensure that none of the part functions, constants, queries, or constructors are from this **Category** or any of its ancestors.
3. Move the adding-new-element-operation further down the lattice, in a similar manner. Thus **cons** and **insert** would not know anything about each other. In this case the automated coercion algorithm would still not be applicable

This is merely one example of many such operators which could require moving.

### 10.9.4 Retyping certain sorts

As in section 10.9.3, this is best illustrated by an example.

In Axiom, the **Category**, **Ring** exports a function,

$$\mathbf{characteristic}: \ () -> \ \mathbf{NonNegativeInteger}$$

(remember that Axiom does not differentiate between sorts and carriers). So imagine the natural ring-epimorphism $\phi : \mathbb{Z}_6 \to \mathbb{Z}_2$. Then the following equation should hold,

$$\mathbf{characteristic}(\phi()) = \phi(\mathbf{characteristic}())$$

yet clearly this is not the case, since $\phi$ must send the **Void** sort to **Void**, and thus the left hand side must equal the characteristic of $\mathbb{Z}_2$, which is 2. Whereas, for the right hand side, we have $\phi(6)$. This $\phi$ is the natural map from $\mathbb{N} \cup \{0\}$ to itself. This is, of course, the identity map, and hence the right hand side has the value 6.

There is clearly something very wrong here. There are two solutions, the first of which is highly unsatisfactory:

1. Stop **characteristic** from being a function. One could persuade it to be an attribute of the **Ring**.
2. Alter the carrier of the return type of **characteristic** from **NonNegativeInteger** to being a type with the same elements, but a different idea of equality.

This is one of many such cases in Axiom.

### 10.9.5 Sorts and their order

We have been discussing Axiom's **Category** inheritance system as if it were a true attempt at modelling order-sorted algebra. There are, however, two areas which are distinctly missing from the Axiom model; these are, the sorts and their order.

There is a distinct confusion in Axiom between sorts and carriers. The author believes that all **Category** definitions (the signatures or theories which specify varieties) should not use genuine types at any point in the signatures, or any other point. These should only occur in the **Domains** of that **Category**.

For example, many **Categories** assume that the only boolean-like type is **Boolean**. Similarly the types **NonNegativeInteger**, **PositiveInteger**, and **Integer** are often "assumed" and are not parameters of a type.

For the types **Boolean**, **NonNegativeInteger**, and **PositiveInteger** this is not normally a complete disadvantage. There are not likely to be other algebras in our system which behave similarly enough to replace these types.

However, there are coercions between **NonNegativeInteger**, **PositiveInteger**, and **Integer** (in the obvious directions). This then imposes some order on the sorts of the type which is not explicitly mentioned.

Some of the sorts of an algebra are defined. % is always the principal sort, and any parameters of the **Category** are there, too. Others, however, are merely mentioned in function prototypes (signatures). All should be listed in a sort-list (and/or sort-lattice, see below).

Neglected sorts normally include those which are carried by the four types mentioned above. However, other types are often neglected too. These include **List** which is often present so that elements of the type may be constructed. (The underlying representation of most types in a Lisp-based system are often lists). More seriously, particular polynomial representations are sometimes present.

There is no real mechanism built into Axiom to order the sorts of a signature (**Category**). The order is implicit in **Categorical** inheritances, such as **CoercibleTo** and **RetractibleTo**, which give information on how the principal sort relates to some other sorts.

I believe that a more sensible way would be to declare a lattice like arrangement with the declaration of a **Category**. Indeed, this would then make the sort-list clear, too, since this is never explicitly defined either.

For example, we could add the syntax, **SortOrder** to be used as follows,

```
ACategory(a:A,b:B,...):Category ==
  with SortOrder{
    a < %;
    % < b;
    ...
  }
  exportedFunction : List A -> %;
...
```

or (better), **SortOrder** could be a **Category** which could be defined as (assuming this is compilable[25]),

[25] According to Peter Broadberry "One problem is that it may not be possible for the compiler to figure

```
SortOrder(ls:List Pair Type):Category == {
  for pr in ls repeat {
    coerce:first pr -> second pr;
  }
}
```

This **for** loop would not build the entire sort lattice for any particular category. This would be better done by the compiler at compile time, on a per-**Category** or per-**Type** basis.

Thus, any **Category** which does not extend **SortOrder** would be a non-order-sorted signature.

### 10.9.6 Altering Axiom's databases

Axiom has many built in databases for looking up comments, functions, attributes, etc. from each **Category** or **Domain**. These are usually text files, with character number keys for faster cross-referencing.

Axiom could have some extra databases to aid the automated coercion algorithm. Specifically, there could be a database containing for each **Category**, a list (of lists) of special functions exported by that algebra, but not its ancestors. This could just be also be a compiled fact about each algebra, but the look-up overhead would be greater.

The restriction on only having functions from that **Category** and not its ancestors' special functions allows for the dynamic nature of Axiom's **Category** system. Without this restriction, altering a **Category** further up the lattice could cause the database information to become outdated for all of its descendants.

### 10.9.7 Conclusion

Axiom behaves very similarly to a language based on order sorted theories and the algebras that model them. However, there are some key areas in which Axiom differs from the mathematical notions.

We summarise these areas in the following table.

---

out exactly what this construct will export" but it might still be implementable.

| Mathematical Notion | Axiom |
|---|---|
| Operator names need not be the same as operator symbols | Operator names are always the same as operator symbols |
| Coercions are homomorphisms and hence act homomorphically on all operators | Coercions need not act like homomorphisms at all. (If it were possible to implement a universal equation checker then Axiom would be alright) |
| A signature depends on its sort-list, and hence a sort-list is part of its definition | Category definitions do not explicitly list their sorts |
| No algebra are mentioned in a signature definition, only sorts | Category definitions do depend on specific Domains |
| The order on the sorts is part of the definition of a signature | Category definitions do not explictly order their sorts |

We have presented methods for addressing all of these differences.

## 10.10 Conclusions

### 10.10.1 Summary

**Category theory and order sorted algebras as the bases for sound strongly typed computer algebra systems**

We have shown why both category theory and order sorted algebra both provide solid models for the type systems found in computer algebra.

In section 10.3.17 we have demonstrated the correlation between a computer algebra type system and category theory.

In section 10.4.48 and 10.5.1 we have shown how a computer algebra type system correlates with order sorted algebra. In section 10.5.10 we extended the notion of order sorted signatures to cover conditional signatures which occur in Axiom.

We have mentioned some of the correspondence between category theory and order sorted algebra (section 10.5.22). We have usually used order sorted algebra as a model. This is because order sorted algebra more readily corresponds to the algebraic inheritance mechanism. This is mainly due to the order on the sorts which is not available in category theory. Also category theory has more difficulty expressing higher order polymorphism.

#### Representation and syntax issues of an order sorted algebra based type system

In section 10.9 we demonstrated various methodologies for extending Axiom's current type system so that it may more closely model order sorted signatures and algebra.

In section 10.9.2 we showed how Axiom's syntax may be extended so that a signature may extend another more than once[26]. We also showed that this syntax could be used to uniquely identify operator names with operator symbols. This then allows operator symbols to differ from operator names.

In section 10.9.5 we discussed how both the sort list and the order on the sorts could be introduced to Axiom's signatures.

Section 10.9.3 commented on how the Axiom signature tree may be altered to allow coercions to act more homomorphically.

Finally, we discussed how Axiom signatures could be compiled to contain extra information which would enhance the speed of the automated coercion algorithm (section 10.9.6).

#### On coherence

In section 10.6 we first stated all the mathematics used by Weber to state and "prove" Weber's coherence conjuecture 10.6.17.

In section 10.6.18 we then showed that this "proof" is not correct. We showed that altering a definition and adding a couple new assumptions that let us prove the coercion theorem 10.6.25.

In section 10.6.26 we then showed that it was possible to relax one of the assumptions which Weber made, and still have a coherent type system. (The extended coherence theorem 10.6.30)

#### The automated coercion algorithm

In section 10.7.1 we provided enough mathematics to show that the automated coercion algorithm 10.7.11 (section 10.7.10) is an algorithm which returns a function which is not only a homomorphism, but a coercion which we defined in definition 10.5.25.

This homomorphism is unique in a coherent type system, which we can guarantee providing we can satisfy all the assumptions of the extended coercion theorem 10.6.30. The homomorphism created may be built from the four basic types of coercion given in the statement of that theorem.

In section 10.8 we showed that a demonstration implementation of this algorith is possible in Axiom.

### 10.10.2 Future work and extensions

We have presented in this thesis the basis for a mathematically sound computer algebra system. We have also shown that it would be possible to implement the automated coercion algorithm in such a system.

At present Axiom comes close but is not fully conforming.

[26] Thus thsi solves an interesting class of multiple inheritance problem.

We have shown how Axiom's syntax could be extended to allow it to model an order sorted algebra system.

Axiom's categories are not always abstract – that is they may contain implementations of functions[27]. This is a good thing as it enforces certain truths about a particular function.

Forcing any important facts by making a class which implements a particular inferface would then ruin any chance of re-attaining multiple inheritance.

[27] The "not equals" fuction is defined to be "not(equals())"

# Chapter 11

# Symmetries of Partial Differential Equations by Fritz Schwarz

This chapter is based on Schwarz [Schw87] "Programming with Abstract Data Types: The Symmetry Package SPDE in Scratchpad".

## 11.1 Symmetries of Differential Equations and the Scratchpad Package SPDE

Symmetry analysis is the only systematic way to obtain solutions of differential equations. Yet it is rarely applied for that purpose and most textbooks do not even mention it. The reason is probably the enormous amount of calculations which is usually involved in obtaining the symmetry group of a given differential equation. Therefore the Scratchpad package SPDE which stands for Symmetries of Partial Differential Equations has been developed which returns the complete symmetry group for a wide class of differential equations automatically. As far as the mathematics is concerned, only those formulas are given which are a prerequisite for the main topic mentioned. The details and many examples may be found in the recent review article on that subject by the author [14].

We consider the most general case of a system of differential equations for an arbitrary number $m$ of the unknown functions $u^\alpha$ which may depend on $n$ arguments $x_i$. These variables are collectively denoted by $u = (u^1, \ldots, u^m)$ and $x = (x_1, \ldots, x_n)$ respectively. We write the system of $N$ differential equations in the form

$$w_\nu(x_i, u^\alpha, u^\alpha_i, u^\alpha_{ij}, \ldots, u^\alpha_{i_1 \ldots i_k}) = 0 \tag{1}$$

for $\nu = 1 \ldots N$ where the notation

$$u^\alpha_{i_1 \ldots i_n} = \frac{\partial^{i_1 + \ldots i_n} u^\alpha}{\partial x_1^{i_1} \ldots \partial x_n^{i_n}}$$

for derivatives has been used. Furthermore it is assumed that the equations (1) are polynomial in all arguments. For $m = n = N = 1$ a single ordinary differential equation is

obtained.

To formulate the condition for the invariance of (1), the infinitesimal generator $U$ is defined by

$$U = \xi_i \frac{\partial}{\partial x_i} + \eta^\alpha \frac{\partial}{\partial u^\alpha} \tag{2}$$

where $\xi_i$ and $\eta^\alpha$ may depend on all dependent and independent variables. Summation over twice occuring indices is always assumed. Greek indices run from 1 to $m$ and latin indices from 1 to $n$. The $k$-th prolongation of $U$ is defined as

$$U^{(k)} = U + \zeta^\alpha_i \frac{\partial}{\partial u^\alpha_i} + \ldots + \zeta^\alpha_{i_1 \ldots i_k} \frac{\partial}{\partial u^\alpha_{i_1 \ldots i_k}} \tag{3}$$

where the functions $\zeta^\alpha_{i_1,\ldots i_k}$ describe the transformation of partial derivatives of order $k$. The $\zeta$'s satisfy the recursion relations

$$\zeta^\alpha_i = D_i(\eta^\alpha) - u^\alpha_s D_i(\xi_s) \tag{4}$$

and

$$\zeta^\alpha_{i_1,\ldots i_k} = D_{i_k}(\zeta^\alpha_{i_1 \ldots i_{k-1}}) - u^\alpha_{i_1,\ldots i_{k-1},s} D_{i_k}(\xi_s) \tag{5}$$

$$D_i = \frac{\partial}{\partial x_i} + u^\alpha_i \frac{\partial}{\partial u^\alpha} + u^\alpha_{ki} \frac{\partial}{\partial u^\alpha_k} + u^\alpha_{kli} \frac{\partial}{\partial u^\alpha_{kl}} \ldots \tag{6}$$

is the operator of total derivation with respect to $x_i$. The system of differential equations (1) is invariant under the transformations of a one-parameter group with the infinitesimal generator (2) if the $\xi$'s and $\eta$'s are determined from the conditions

$$U^{(k)} \omega_\nu = 0 \quad \text{when all } \omega_\nu = 0. \tag{7}$$

Under the constraints for the $\omega_\nu$ which have been mentioned above, the left hand side of (7) is a polynomial in all its variables. Because the derivatives of the $u$'s do not occur as arguments of the $\xi$'s and the $\eta$'s, it has to be decomposed with respect to these derivatives and the coefficients are equated to zero. The resulting set of equations is the determining system the general solution of which determines the full symmetry group of (1). Starting from a certainset of simplification rules, a solution algorithm has been designed which is described in detail in a separate article [Schw85]. The implementation of this algorithm forms the main part of the package SPDE wich comprises about 1500 lines of Scratchpad code.

Due to its size, the crucial part of the implementation is to identify a set of datatypes such that a modularization is obtained. This is not a single step process but involves a lot of trial and error and also some backtracking. The basic building block for these new datatypes in the Scratchpad domain SMP(R,VarSet) which abbreviates Sparse Multivariate Polynomial in the VarSet over a ring R. The latter may be e.g. the integers, the rational numbers or another polynomial ring over some set of variables. There are three basic variables distinguished which occur in equations (1) to (7). These are the $x_i$ and $u^\alpha$, the derivatives $u^\alpha_{i_1,\ldots i_k}$ and the differential operaotrs $\partial/\partial x_i$ and $\partial/\partial u^\alpha$. They are represented in Scratchpad Symbols of the type DEVAR, DER, and DO respectively. Furthermore there are the $\xi$'s and the $\eta$'s together with the $c_k$'s which are introduced by the solution algorithm. These variables of the type LDFV are also Scratchpad Symbols. However they are special in the sense that they carry dependencies with them which may change while the solution algorithm proceeds. The bookkeeping for these dependencies is organized in terms of a Scratchpad association list.

For reasons that will become clear soon it is advantageous to introduce still another kind of variables of type DK which represent the derivatives of the previously introduced variables LDFV. They do not correspond straightforwardly to a Scratchpad system type.

Out of these variables all quantities which occur may be built up in terms of SMP's as follows. The differential equations themselves are considered as polynomials in the derivatives of $u^\alpha_{i_1,\ldots i_k}$ with coefficients which are polynomials in the $x_i$ and $u^\alpha$ over the rationals, i.e. their type is SMP(SMP(RN,DEV),DER). The $\zeta$'s are linear polynomials in the $\xi$'s, the $\eta$'s and derivatives thereof with coefficients which are polynomials in the derivates $u^\alpha_{i_1,\ldots i_k}$ over the integers, i.e. the appropriate type is SMP(SMP(I,DER),DK). The equations of the determining system are obtained by decomposing the left hand side of (7) with respect to the derivatives $u^\alpha_{i_1,\ldots i_k}$. The resulting equations of the determining system are linear polynomials in the DK's with coefficients which are polynomials in the variables $x_i$ and $u^\alpha$ over the rational numbers. They are denoted by the new type LDF. The symmetry generators which are obtained from the solution of the determining system are linear polynomials in the differential operators $\partial/\partial x_i$ and $\partial/\partial u^\alpha$. Depending on whether or not there is a functional dependency involved in the final solution their coefficients are LDF's or polynomials in the DEV's over the rational numbers respectively. So the two kinds of generators are SMP(LDF,DO)'s or SMP(SMP(RN,DEV),DO)'s for which the two type CSG and DSG respectively are introduced. The complete set of domains of the symmetry package SPDE is listed below where the full names are also given.

| Abbreviation | Full Name | Scratchpad Datatype |
|---|---|---|
| SPDE | SymmetricPartialDifferentialEquation | Package |
| CSG | ContinuousSymmetryGenerator | SMP(LDF,DO) |
| DSG | DiscreteSymmetryGenerator | SMP(SMP(RN,DEV),DO) |
| DS | DeterminingSystem | List List LDF |
| LDF | LinearDifferentialForm | SMP(SMP(RN,DEV),DK) |
| DK | DifferentialKernel | New Domain |
| LDFV | LDFVariable | Symbol |
| DE | DifferentialEquation | SMP(SMP(RN,DEV),DER) |
| DER | Derivative | Symbol |
| DO | DifferentialOperator | Symbol |
| DEV | DEVariable | Symbol |

An abstract data type is realized in Scratchpad in terms of a domain constructor. As an example in the following figure the specification of the domain DifferentialKernel is shown. According to the principles of Scratchpad, there is a public- or category part Cat and a private part Tar. The category part Cat defines the outside view. It consists of the syntax specification for the exported functions in terms of its modemaps and the semantic part in which the meaning of these functions is specified. A modemap for a function is a statement which determines the number and the types of its arguments and the type of the object it returns. Instead of a so called axiomatic or algebraic specification a concise and precise description of the action of each function in plain English is preferred. It is included as a comment in the domain constructor. Analogously the private part Tar specifies the syntax and the semantics of the internal functions. The difference between the public- and the private part should be noted. In the former there is no mention whatsoever of the internal representation of these objects in terms of certain records. The semantic specification is mostly given in mathematical terms. On the contrary, in the private part the internal representation of these quantities is established. The terms which are used in its specification are typical for the Scratchpad system.

The function randDK is a random generator for DK's. Its two arguments specify the values of rn and n. It works according to the following algorithm. At first a variable of the type LDFV is created by calling the random generator from the corresponding domain LDFV. Then a random integer between 0 and 5 is generated which specifies an upper bound for the total order of the kernel to be returned. Finally in a loop the derivatives with respect to the various arguments are determined by generating random integers between 0 and 5. The loop terminates if the total order is exceeded. In this way DK's are obtained which cover fairly uniformly the parameter space which is expected to be relevant for applications of the full package, including special cases like e.g. 0-th order derivatives. This random generator for DK's is called by the test program testDK and by test programs for other domains like e.g. LDF. The details of this testing process will be discussed later in this Section. The domain constructors for the other datatypes are similarly organized. The reason for choosing DK as an example has been that it is short enough to be reproduced on a single page but still contains all the relevant details.

```
)abbreviate domain DK DifferentialKernel
DifferenetialKernel: Cat == Tar where
  I ==> Integer
  DEV ==> DEVariable
  LDFV ==> LDFVariable
  VAR ==> Record(Var: DEV, Ord: Integer)
  Cat == OrderedSet with
    funDK: $ -> LDFV            -- function argument
    varDK: $ -> List DEV        -- derivative variables
    zeroDK: $ -> Boolean        -- true if derivative is 0
    newKD: LDFV -> $            -- creates DK of 0th order from argument
    difDK: ($,DEV) -> $         -- derivative w.r.t 2nd argument
    intDK: ($,DEV) -> $         -- integration w.r.t 2nd argument
    ordDK: $ -> I               -- total order of derivative
    ordDK: ($,DEV) -> I         -- order w.r.t 2nd argument
    oneDK: List $ -> LIst $     -- list elements occuring once
    randDK: (I,I) -> $          -- generates random DK
    testDK: (I,I,I,I) -> Void   -- test program
    coerce: $ -> E              -- print function
  Tar == add
    Rep := Record(fn: LDFV, args: List VAR)
    mkDfL: (List VAR,DEV) -> List VAR
        -- creates record VAR for derivative
    mkIntL: (List VAR,DEV) -> List Var
        -- creates record VAR for integral
    creDK: (LDFV,List VAR) -> $
        -- creates DK from LDFV and record VAR
    VarDK: $ -> List VAR
        -- returns record VAR of argument
```

After the various modules which build up the full package SPDE have been established their mutual relations have to be investigated. All dependencies between the modules are most clearly seen from the structure chart which is shown in Figure 4. It makes obvious the hierarchical order between the various modules which is based on the datatypes. The tree-like appearance reflects the most valuable feature of the design, i.e. the partial independence among the modules. For example, those at the bottom which belong to the level of symbols and kernels are almost completely independent from each other. The same is true at the next level of the SMP s. Only at the uppermost level a strong interconneetion is established

among the modules of the full package due to the operations of the module SPDE. This is not surprising however since it is the task of that latter module to organize the cooperation within the package. This becomes clear already from the fact that SPDE is a Scratchpad package constructor whereas all other modules are domain constructors. To emphasize this significant difference the interconnections between modules have been marked with heavy nes whereas for all dependencies on the package constructor SPDE thin lines are applied.

# Chapter 12

# Primality Testing Revisited by James Davenport

This is from Davenport [Dave92]

It is customary in computer algebra to use the algorithm presented by Rabin [Rabi80] to determine if numbers are prime (and primes are needed throughout algebraic algorithms). As is well known, a single iteration of Rabin's algorithm, applied to the number $N$, has probability at most of 0.25 of reporting "$N$ is probably prime", when in fact $N$ is composite. For most $N$, the probability is much less than 0.25. Here, "probability" refers to the fact hat Rabin's algorithm begins with the choice of a "random" seed $x$, not congruent to 0 modulo $N$. In practice, however, true randomness is hard to achieve, and computer algebra systems often use a fixed set of $x$ – for example, Axiom release 1 uses the set

$$\{3, 4, 7, 11, 13, 17, 19, 23, 29, 31\} \tag{1}$$

As Pomerance *et al.* [Pome80] point out, there is some sense in using primes as the $x$-values: for example the value $x = 4$ gives no more information than the value $x = 2$, and the value $x = 6$ can only give more information than 2 or 3 under rare circumstances (in particular, we need the 2-part of the orders of 2 and 3 to differ, but be adjacent). By Rabin's theorem, a group-theoretic proof of which is given in Davenport and Smith [Dave87], 10 elements in the set gives a probability less than 1 in $10^6$ of giving the wrong answer. In fact, it is possible to do rather better than this: for example Damgard and Landrock [Damg91] show that, for 256-bit integers, six tests give a probability of less than $2^{-51}$ of giving the wrong answer.

Nevertheless, given any such fixed set of $x$ values, there are probably some composite $N$ for which all the $x$ in the set report "$N$ is probably prime". In particluar Jaeschke [Jaes91] reports that the 29-digit number

$$56897193526942024370326972321 = 137716125329053 \cdot 413148375987157$$

has this property for the set (1). For brevity, let us call this number $J$ – "the Jaeschke number", and its factors $J_1$ and $J_2$ respectively. Now

$$J = 1 + 2^5 \cdot 1778037297716938261572717885$$

so Rabin's algorithm will begin by raising each element of (1) to the power

177803729771693826157271788 5 (modulo $J$), thus getting

$$\begin{array}{rcrcl}
3 & \to & 1 & \overset{\text{squaring}}{\to} & 1 \\
5 & \to & 419906106813101271408407 4012 & \overset{\text{squaring}}{\to} & J-1 \\
7 & \to & 402496834176927012700278671 21 & \overset{\text{squaring}}{\to} & J-1 \\
11 & \to & 402496834176927012700278671 21 & \overset{\text{squaring}}{\to} & J-1 \\
13 & \to & 526981324588110116562428983 09 & \overset{\text{squaring}}{\to} & J-1 \\
17 & \to & 419906106813101271408407 4012 & \overset{\text{squaring}}{\to} & J-1 \\
19 & \to & 402496834176927012700278671 21 & \overset{\text{squaring}}{\to} & J-1 \\
23 & \to & 166475101092493231002991052 00 & \overset{\text{squaring}}{\to} & J-1 \\
29 & \to & 402496834176927012700278671 21 & \overset{\text{squaring}}{\to} & J-1 \\
31 & \to & 1 & \overset{\text{squaring}}{\to} & 1
\end{array}$$

Hence, for all these $x$-values, Rabin's algorithm will say "$J$ is probably prime", since we arrive at a value of 1 in our repeated squaring, either directly ($x = 3$ and $x = 31$) or via $J - 1$. However, this table indicates (to the suspicious human eye) two things.

(A) The first is that $-1$ appears to have four square roots modulo $J$, viz.

$$419906106813101271408407 4012$$

$$402496834176927012700278671 21$$

$$166475101092493231002991052 00$$

$$526981324588110116562428983 09$$

This contradicts Lagrange's theorem, so $J$ cannot be prime.

(B) The second is that, if $J$ were prime, we would expect about half of the elements of (1) to be quadratic non-residues, and hence to need five squarings to reach 1 (4 to reach $J - 1$), about a quarter to be quadratic residues, but quartic non-residues, hence needing three squarings to reach $J - 1$, and only an eighth to be octic residues or better, and to reach $J - 1$ in at most one squaring. Hence, if $J$ were prime, we have observed an event of probability $(1/8)^{10}$ – less than 1 in $10^9$.

Much of thie paper is taken up with a detailed exploration of these observations and their generalisations. We observe that, at least in principle, we are only concerned with the problem of testing relatively large numbers: numbers less than $25 \cdot 10^9$ are covered by Pomerance *et al.* [Pome80].

## 12.1 Rabin revisited

Throughout this paper, we assume that all integers to be tested for primality are positive and odd. We use the standard notation

$$\phi(n) = |\{x : 0 < x \leq n; \gcd(x, n) = 1\}|$$

from which the Chinese Remainder Theorem gives (here and always, we assume in such formulae that the $p_i$ are distinct primes)

$$\phi(\prod p_i^{\alpha_i}) = \prod p_i^{\alpha_i - 1}(p_i - 1)$$

In addition, we introduce

$$\hat{\phi}(\prod p_i^{\alpha_i}) = \text{lcm}(p_i^{\alpha_i - 1}(p_i - 1))$$

Clearly $\hat{\phi}(n) \mid \phi(n)$

The Fermat-Euler Theorem states that, if $\gcd(x, n) = 1$, then $x^{\phi(n)} \equiv 1 \pmod{n}$. This leads to what might be called the Fermat primality test: pick some $x \not\equiv 0 \pmod{n}$ and compute $x^{n-1} \pmod{n}$. If this is not 1, then $n - 1 \neq \phi(n)$, so $n$ cannot be prime. If $x^{n-1} \equiv 1 \pmod{n}$, but $n$ is not prime, we say that $n$ is a pseudoprime($x$). All composite numbers are pseudoprime(1).

However, the Chinese Remainder Theorem implies a stronger result than the Fermat-Euler Theorem, viz. the following.

**Lemma 1.** *if $gcd(x, n) = 1$, then $x^{\phi(n)} \equiv 1$ (mod $n$). Furthermore, $\hat{\phi}(n)$ is minimal with this property.*

A non-prime number $N$ for which $\hat{\phi}(N) \mid N - 1$ is called a Carmichael number. Any Carmichael number has to have at least three prime factors. (If $pq$ were a Carmichael number, then $pq \equiv 1 \pmod{p-1}$, so $q \equiv 1 \pmod{p-1}$ and $q \geq p$. Similarly, $p \equiv 1 \pmod{q-1}$ and $p \geq q$. So $p = q$, but $\hat{\phi}(p^2) = p(p-1)$, which can never divide $p^2 - 1$.) These numbers, of which we now know that there are infinitely many [Alford *et al.* [Alfo92]], are the bane of the Fermat primality test, since, unless we hit on an $x$ with $\gcd(x, n) \not\equiv 1$, we will always have $x^{N-1} \equiv 1 \pmod{N}$.

Hence we need a stronger test: Rabin's test, which is finer than the Fermat test since, instead of computing $x^{N-1}$, it writes $N - 1 = 2^k - l$ with $l$ odd, and then considers each of $x^1, x^{2l}, \ldots, x^{2^k l}$, (each obtained by squaring the previous one, and all computed modulo $N$). If the last is not 1, we have a non-prime by the Fermat test. If the first is 1 or $N - 1$, we know nothing and say "N is probably prime". If, however, the first 1 is preceded by a number other than $N - 1$, we can assert that $N$ is definitely composite, since we have found a square root of unity other than 1 and $N - 1$.

Another way of seeing the difference between Rabin's test and the Fermat test is to say that we are analysing the 2-part of the order of $x$ modulo $N$ more carefully. We reply "$N$ is definitely not prime" if the order of $x$ has different 2-parts modulo different factors of $N$.

Our starting code for Axiom's implementation (slightly modified from that distributed with Axiom release 1, in particular to split out the auxiliary function `rabinProvesComposite`, but using the same algorithm) is

Original Code

```
[ 1] prime? n ==
[ 2]    n < two => false
[ 3]    n < nextSmallPrime => member?(n, smallPrimes)
[ 4]    not one? gcd(n, productSmallPrimes) => false
[ 5]    n < nextSmallPrimeSquared => true
[ 6]    nm1:=n-1
[ 7]    q := (nm1) quo two
[ 8]    for k in 1.. while not odd? q repeat q := q quo two
[ 9]    -- q = (n-1) quo 2**k for largest possible k
[10]    mn := minIndex smallPrimes
[11]    for i in mn+1..mn+10 repeat
[12]      rabinProvesComposit(smallPrimes i,n,nm1,q,k) => return false
[13]    true
```

```
[14]
[15] rabinProvesComposite(p,n,mn1,q,k) ==
[16]    -- nm1 = n-1 = q*2**k; q odd
[17]    -- probability false for n composite is < 1/4
[18]    -- for most n this probability is much less than 1/4
[19]    t := powmod(p, q, n)
[20]    -- neither of these cases tells us anything
[21]    if not (one? t or t = nm1) then
[22]       for j in 1..k-1 repeat
[23]           t := mulmod(t, t, n)
[24]           one? t => return true
[25]           -- we have squared somethiong not -1 and got 1
[26]           t = mn1 =>
[27]               leave
[28]       not (t = nm1) => return true
[29]    false
```

where we have numbered the lines for ease of refernce. **I** is the datatype of $n$, and can be thought of as being the integers. `smallPrimes` is a list of primes up to 313, and `nextSmallPrime` is therefore 317.

## 12.2 Non-square-free numbers

If Rabin's algorithm is handed a number $N$ with a repeated prime factor $p^k$, then the factor of $p^{k-1}$ in $\hat{\phi}(N)$ will certainly be coprime to $N-1$. This means that we will return "$N$ is definitely not prime" unless we use an $x$-value which is actually a perfect $p^{k-1}$-st power – an event with probability $1/p^{k-1}$. This probability is less than 0.25 except in the case $p=3$, $k=2$, when we can calculate explicitly that the probability of incorrectly saying "$N$ is probably prime" is exactly 0.25 in the case $N=9$.

In the implementation given above, then test at line [ 4] ensures that $N$ has no factors less than 317, and, *a fortiori*, no such repeated factors. Hence the probability that an $x$-value would be a perfect $p$-th power is at most $1/317$. This compares favourably with some of the probabilities that will be analysed later, and shows the practical utility of this preliminary test.

## 12.3 Jaeschke analysed

Let us analyse the number $J$ more closely. To begin with, both $J_1$ and $J_2$ are prime. These numbers can be written as

$$J_1 = 1 + 2^2 \cdot 3^2 \cdot 829 \cdot 4614533083$$

$$J_2 = 1 + 2^2 \cdot 3^3 \cdot 829 \cdot 4614533083$$

$$J = 1 + 2^5 \cdot 3^2 \cdot 5 \cdot 11 \cdot 59 \cdot 829 \cdot 34849 \cdot 456679 \cdot 4614533083$$

$J$ is not a Carmichael number, but it is "fairly close", since it is only the factor of $3^3$, rather than $3^2$, in $J_2 - 1$ which prevents it from being so. In addition, $J$ is a product of two primes, of the form $(K+1)\cdot(rK+1)$ (with $r=3$) – a form observed by Pomerance *et al.* [Pome80] to account for nearly all pseudoprimes.

Why does Rabin's test (using the primes (1)) think that $J$ is prime? To begin with, all the primes in the set (1) are actually perfect cubes modulo $J_2$, so their orders divide $(J_2 - 1)/3$, and hence $J - 1$. Put another way, $J$ is a pseudoprime($p$) for all the $p$ in (1): these 10 primes all cause the Fermat test to be satisfied. Assuming that $3 \mid p - 1$, 1/3 of non-zero congruenece classes are perfect cubes modulo $p$.

For $J$ to pass Rabin's test, we must also ensure that, for every $p$ in (1), the 2-part of the order of $p$ modulo $J_1$ is equal to the 2-part of the order of $p$ modulo $J_2$. 3 and 3l are both quadratic residues modulo both $J_1$ and $J_2$, whilst the other primes are all non-residues. For the non-residues, the 2-part is maximal, viz $2^2$ modulo both these factors, so these eight primes all cause $J$ to pass Rabin's test, as well as Fermat's. 3 and 3l are, in fact, not only quadratic residues, but also quartic residues for both $J_1$ and $J_2$, so their orders have 2-part $2^0$, and hence also cause $J$ to pass Rabin's test.

Since $J_2 \equiv J_1 \equiv 1 \pmod 4$, the quadratic character $(a \mid J_i) = (J_i \mid a)$, and so depends only on the value of $J_i \pmod a$ (in general, one might have to work modulo $4a$). $J_2 = 3J_1 - 2$, so the two are not independent, but we would expect 1/4 of congruence classes of $J_1 \pmod a$ to make $a$ a non-residue for both $J_1$ and $J_2$. Another 1/4 would have $a$ a quadratic residue for both, but it would then be necessary to investigate quartic properties, and so on. For a given $a$, asymptotically, about 1/3 of the values of $J_1$ will arrange that the quadratic, quartic, octic etc. characters of $a$ module $J_1$ and $J_2$ are compatible with passing Rabin's tightening of the Fermat test.

What are the implications of this for an $n$-step Rabin algorithm, if our opponent, the person who is trying to find a composite $N$ such that our use of Rabin's algorithm says "$N$ is probably prime", chooses $N = M_1 \cdot M_2$, with $M_1$, $M_2$ prime and $M_2 - 1 = 3(M_1 - 1)$ (and hence $M_1 \equiv 1 \pmod 3$, otherwise $x = 3$ will fail Rabin's test)? Each prime $p$ we use forces the condition that $p$ should be a perfect cube module $M_2$ – satisfied about 1/3 of the time. In addition, the quadratic characters of $p$ module $M_1$ and $M_2$ must be compatible – at best, with $M_1 - 1 \equiv 2 \pmod 4$, this happens 1/3 of the time on average. Hence each $p$ we use imposes constraints satisfied about 1/9 of the time (assuming independence, which seems in practice to be the case). So we might expect to find a "rogue" number with $M_1$ about $9^n$, and so $N$ is about $9^{2n}$, which is $10^{19}$ if $n = 10$. However, we also have to insist that $M_1$ and $M_2$ are prime, which reduces our chance of finding a rogue pair quite considerably – roughly by 1/22 for each of $M_1$ and $M_2$, which would give us an estimated "time to find a rogue value" of $5 \cdot 10^{21}$. We can, in fact, be surprised that $J$ is as large as it is – perhaps a smaller value exists.

## 12.4 Roots of −1

Here we look at observation (A) above – that a suspicious human being would observe more than two square roots of −1, and hence deduce that $J$ was not prime, irrespective of the details of Rabin's algorithm. This is certainly true – how programmable, and how widely applicable, is it? A global (to `prime?` and `rabinProvesComposite`) variable `rootsMinus1` is added, whose type is a `Set` of `I`. The code now reads:

```
                        "Roots of -1" Modifications

[ 1] prime? n ==
[ 2]    n < two => false
[ 3]    n < nextSmallPrime => member?(n, smallPrimes)
```

```
[ 4]    not one? gcd(n, productSmallPrimes) => false
[ 5]    n < nextSmallPrimeSquared => true
[ 6]    nm1:=n-1
[ 7]    q := (nm1) quo two
[ 8]    for k in 1.. while not odd? q repeat q := q quo two
[ 9]    -- q = (n-1) quo 2**k for largest possible k
[10]    mn := minIndex smallPrimes
[10g]   rootsMinus1 := [] -- the empty set
[11]    for i in mn+1..mn+10 repeat
[12]      rabinProvesComposit(smallPrimes i,n,nm1,q,k) => return false
[13]    true
[14]
[15] rabinProvesComposite(p,n,mn1,q,k) ==
[16]    -- nm1 = n-1 = q*2**k; q odd
[17]    -- probability false for n composite is < 1/4
[18]    -- for most n this probability is much less than 1/4
[19]    t := powmod(p, q, n)
[20]    -- neither of these cases tells us anything
[21]    if not (one? t or t = nm1) then
[22]       for j in 1..k-1 repeat
[22a]          oldt := t
[23]           t := mulmod(t, t, n)
[24]           one? t => return true
[25]           -- we have squared somethiong not -1 and got 1
[26]           t = mn1 =>
[26a]              rootsMinus1:=union(rootsMinus1,oldt)
[26b]              # rootsMinus1 > 2 => return true
[27]               leave
[28]       not (t = nm1) => return true
[29]    false
```

These changes certainly stop the algorithm from returning "N is probably prime" on the Jaeschke number, and do not otherwise alter the correctness of the algorithm, so might as well be incorporated. They only take effect when $k > 1$, since only then is the loop at [22] onwards executed.

If $k > 1$ then these changes certainly may be executed. But if all the prime factors $p_i$ of $N$ have small 2-part in $\phi(p_i)$, in particular if the 2-part of $\hat{\phi}(N) = 2^1$, then these changes will not take effect (but those proposed in the next section will). In general it is hard to analyse the precise contribution of these changes, other than to be certain that it is never negative.

## 12.5 The "maximial 2-part" test

Here we attempt to generalise observation (B) above. Let us suppose that $N$ is still the composite number that we wish to prove is composite, and that $N = \prod_{i=1}^{n} p_i$ with the $p_i$ distinct. Write $N = 1 + 2^k l$ with $l$ odd, and $p_i = 1 + 2^{k_i} l_i$, with $l_i$ odd. Clearly $k \geq \min_i k_i$, if $N$ were prime, we would know that half the residue classes modulo $N$ were quadratic non-residue, and hence we would expect half the $x$-values chosen to have 2-order $k$. Conversely, if all the $k_i$ were equal to each other and to $k$, we would expect $X$ to be a quadratic non-residue about half the time *with respect to each* $p_i$, and so about 1 in $2^n$ of the $x$-values will have maximal 2-rank.

One very simple variant on this test that can be imposed is to insist that, before deciding

that "$N$ is probably prime", we actually observe an element of 2-order $k$. If $N$ actually were prime, we would have a chance of 1023/1024 of observing this before finishing the loop starting on line [11], so this test is extremely unlikely to slow down the performance of the system on primes. On non-primes, it may slow us down, but increases the change of our giving the "correct" answer.

We need a global (to `prime?` and `rabinProvesComposite`) variable `count2Order` whose type is a `Vector` of `NonNegativeIntegers`. This variable is used to count the number of elements of each 2-order.

"Maximal 2-part" Modifications

```
[ 1] prime? n ==
[ 2]    n < two => false
[ 3]    n < nextSmallPrime => member?(n, smallPrimes)
[ 4]    not one? gcd(n, productSmallPrimes) => false
[ 5]    n < nextSmallPrimeSquared => true
[ 6]    nm1:=n-1
[ 7]    q := (nm1) quo two
[ 8]    for k in 1.. while not odd? q repeat q := q quo two
[ 9]    -- q = (n-1) quo 2**k for largest possible k
[10]    mn := minIndex smallPrimes
[10g]   rootsMinus1 := [] -- the empty set
[10h]   count2Order := new(k,0) -- vector of k zeros
[11]    for i in mn+1..mn+10 repeat
[12]      rabinProvesComposit(smallPrimes i,n,nm1,q,k) => return false
[12e]     currPrime:=smallPrimes(mn+10)
[12f]     while count2Order(k) = 0 repeat
[12g]           currPrime := nextPrime currPrime
[12h]           rabinProvesComposite(currPrime,n,nm1,q,k) => false
[13]    true
[14]
[15] rabinProvesComposite(p,n,mn1,q,k) ==
[16]    -- nm1 = n-1 = q*2**k; q odd
[17]    -- probability false for n composite is < 1/4
[18]    -- for most n this probability is much less than 1/4
[19]    t := powmod(p, q, n)
[19a]   if t=mn1 then count2Order(1):=count2Order(1)+1
[20]    -- neither of these cases tells us anything
[21]    if not (one? t or t = nm1) then
[22]       for j in 1..k-1 repeat
[22a]          oldt := t
[23]          t := mulmod(t, t, n)
[24]          one? t => return true
[25]          -- we have squared somethiong not -1 and got 1
[26]          t = mn1 =>
[26a]              rootsMinus1:=union(rootsMinus1,oldt)
[26b]              # rootsMinus1 > 2 => return true
[26c]              count2Order(j+1):=count2Order(j+1)+1
[27]              leave
[28]       not (t = nm1) => return true
[29]    false
```

We currently collect more information than we use. Again, this modification to the Rabin algorithm proves that the Jaeschke number is not prime.

## 12.6 How would one defeat these modifications?

It is all very well to propose new algorithms, an demonstrate that they are "better" than the old ones, but might they really have loop-holes just as large? The "maximal 2-part" requirement defeats a whole family of pseudoprimes – all those of the form $(K+1)\cdot(rK+1)$ with $r$ odd, since then $N-1$ has a higher 2-part than $\hat{\phi}(N)$. This test is therefore useful in general, and defeats any straight-forward generalisation of the Jaeschke number to larger sets of $x$.

There are various possible constructions which these modifications do not defeat. We could make our pseudoprime $N$ take the form $(K+1)\cdot(6K+1)$ with $K \equiv 2 \pmod 4$. Then the 2-part of $\hat{\phi}(N)$ would be $2^2$, whereas that of $N-1$ would be $2^1$ (and so the "roots of -1" enhancement would never operate). A value $x$ would pass Rabin's test, with the "maximal 2-part" enhancement, if it were

(1) a cubic residue modulo $6K+1$

(2) a quadratic residue modulo $6K+1$

(3) a quartic non-residue modulo $6K+1$

(4) a quadratic non-residue modulo $K+1$

On average, one $K$-value in 24 will have these properties for a fixed $x$.

A value $x$ would also pass Rabin's test, but would not contribute to the "maximal 2-part", if it were

(1) a cubic residue modulo $6K+1$

(2) a quadratic residue modulo $6K+1$

(3) a quartic residue modulo $6K+1$

(4) a quadratic residue modulo $K+1$

Again, on average, one $K$-value in 24 will have these properties for a fixed $x$.

We note, therefore, that we might expect 50% of $x$-values causing $N$ to pass Rabin's test to have 2-part $2^1$ and 50% to have 2-part $2^0$; the same distribution as for a prime value of $N$ (with $k=1$). If we use $n$ different $x$-values, we might expect $K$ to have to be of the order of $12^n$, and $N$ to be of the order of $144^n$. In addition, both $K+1$ and $6K+1$ have to be prime. For $n=10$, the probability of this is about $1/25$, so we might expect to find such an $N$ at around $2\cdot 10^{24}$.

## 12.7 Leech's attack

Leech [Leec92] has suggested an attack of the form $N=(K+1)\cdot(2K+1)\cdot(3K+1)$. If the three factors are prime (which incidentally forces $K=2$, a case we can discard, or $K \equiv 0 \pmod 6$), then these numbers are certainly Carmichael, and hence a good attack on the original version of Rabin's algorithm. Indeed, almost 25% of seed values will yield the result "$N$ is probably prime".

Fortunately, we are saved by the "maximal 2-part" variant. Suppose $K=3\cdot n\cdot 2^m$ with $n$ odd (and $m$ at least 1). Then the maximal 2-part we can actually observe is $2^m$, whereas

$$N-1 = 162n^3(2^m)^3 + 90n^2(2^m)^2 + 18n2^m$$

which is divisible by $2^{m+1}$. Hence we will never observe an element of the maximal 2-part, and the loop at line [12f] will run until a counter-example to primality is found.

In fact, if $m = 1$, $N-1$ is divisible by 8, and if $m > 1$, $N-1$ is divisible by $2^{m+1}$, which is at least 8. Hence the "roots of -1" test also acts, and reduces the probability of passing the modified Rabin well below 25%.

Other forms of attack are certainly possible, e.g. taking $N = (K+1)\cdot(3K+1)\cdot(5K+1)$. Here the "maximal 2-part" does not help us, since the 2-part of $N-1$ is equal to that of $\hat{\phi}(N)$. However these numbers are not generally Carmichael, only "nearly Carmichael", since 5 does not divide $N-1$. Hence we would need to insist that all our seed values were quintic residues modulo $5K+1$ as well as having the same 2-part modulo all the factors, and so on. These more complex families seem to create more problems for the inventor of counter-examples, so we can probably say that taking one prime for every factor of 100 in $N$ probably makes the systematic construction of counter-examples by this technique impossible.

However, if we also force $K \equiv 12 \pmod{30}$, Leech [Leec92] has pointed out that $N$ is Charmichael. By construction, the factors are congruent to each other, and to their product, modulo 12, so the quadratic characters of 3 modulo the different factors are compatible. In fact we also need $K \equiv 0 \pmod{7}$, since $K \equiv 1, 3, 5$ gives incompatible quadratic characters for 7 modulo the different factors, and $K \equiv 2, 4, 6$ gives non-prime factors. However, the three factors are congruent respectively to 3, 2 and 1 modulo 5, and so 5 will be a quadratic non-residue modulo the first two factors, but a residue modulo the last, hence ensuring that Rabin's algorithm with $x = 5$ always says "$N$ is certainly composite".

## 12.8 The $(K+1)\cdot(2K+1)$ attack

This attack has been used recently by Arnault [Arna91] to defeat the set of $x$-values

$$\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\}$$

The number

$$1195068768795265792518361315725116351898245581 =$$

$$48889032896862784894921 \cdot 24444516448431392447461$$

passes all these tests. In effect, the requirement is that $x$ be a quadratic residue modulo $2K+1$, and that the quadratic character of $x$ modulo $K+1$ should equate to the quartic character of $x$ modulo $2K+1$. These conditions are satisfied for approximately 25% of $K$-values. In addition, of course, $K+1$ and $2K+1$ must be prime. It would almost certainly be possible to construct a much smaller number than Arnault's, with the same properties – he fixed the congruences classes he was considering; for example he chose one class modulo 116, rather than examining all 29 satisfactory classes.

This form of attack is particularly worrying, since it is much easier to use than the other attacks in the previous sections. Indeed, one should probably consider $\log_4 N$ values $x$ to test a number $N$ if one is to defend against this attack. Fortunately, we have a simpler defence; we can check explicitly if the number we are given is of the form.

It is worth noting that Damgard and Landrock [Damg91] prove the following.

**Theorem** *If $N$ is an odd composite number, such that $N$ is not divisible by 3, and more than 1/8 of the x-values yield "$N$ is probably prime" then one of the following holds*

- *$N$ is a Carmichael number with precisely three prime factors*

- $3N + 1$ *is a perfect square*
- $8N + 1$ *is a perfect square*

$8(K + 1) \cdot (2K + 1) + 1 = (4K + 3)^2$, so this attack is a special case of the above theorem. There seems no reason not to test both the exceptional conditions in the Damgard-Landrock Theorem – such numbers are always composite, except for trivial cases ruled out by lines before [ 5].

"Damgard-Landrock" Modifications

```
[ 1] prime? n ==
[ 2]    n < two => false
[ 3]    n < nextSmallPrime => member?(n, smallPrimes)
[ 4]    not one? gcd(n, productSmallPrimes) => false
[ 5]    n < nextSmallPrimeSquared => true
[ 6]    nm1:=n-1
[ 7]    q := (nm1) quo two
[ 8]    for k in 1.. while not odd? q repeat q := q quo two
[ 9]    -- q = (n-1) quo 2**k for largest possible k
[10]    mn := minIndex smallPrimes
[10g]   rootsMinus1 := [] -- the empty set
[10h]   count2Order := new(k,0) -- vector of k zeros
[11]    for i in mn+1..mn+10 repeat
[12]      rabinProvesComposit(smallPrimes i,n,nm1,q,k) => return false
[12a]     import IntegerRoots(I)
[12b]     q > 1 and perfectSquare?(3*n+1) => false
[12c]     ((n9:=n rem (9::I))=1 or n9 = -1) and perfectSquare(8*n+1) => false
[12d]     -- Both previous tests fro Damgard & Landrock
[12e]     currPrime:=smallPrimes(mn+10)
[12f]     while count2Order(k) = 0 repeat
[12g]           currPrime := nextPrime currPrime
[12h]           rabinProvesComposite(currPrime,n,nm1,q,k) => false
[13]    true
[14]
[15] rabinProvesComposite(p,n,mn1,q,k) ==
[16]    -- nm1 = n-1 = q*2**k; q odd
[17]    -- probability false for n composite is < 1/4
[18]    -- for most n this probability is much less than 1/4
[19]    t := powmod(p, q, n)
[19a]   if t=mn1 then count2Order(1):=count2Order(1)+1
[20]    -- neither of these cases tells us anything
[21]    if not (one? t or t = nm1) then
[22]       for j in 1..k-1 repeat
[22a]          oldt := t
[23]           t := mulmod(t, t, n)
[24]           one? t => return true
[25]           -- we have squared somethiong not -1 and got 1
[26]           t = mn1 =>
[26a]              rootsMinus1:=union(rootsMinus1,oldt)
[26b]              # rootsMinus1 > 2 => return true
[26c]              count2Order(j+1):=count2Order(j+1)+1
[27]               leave
[28]       not (t = nm1) => return true
[29]    false
```

## 12.9 Conclusions

It is certainly possible to draw more information from a fixed set of $x$-values than Rabin's original algorithm does, and we have explained two ways of doing this. While we have not yet constructed a number that defeats our enhanced version of Rabin's algorithm, it should certainly be possible to do so, if the set of $x$-values is fixed. In general, the number of primes used should be proportional to $\log N$, and we have made some suggestions as to what the constant of proportionality should be. A better constant of proportionality can be used if we test explicitly for numbers of the form $(K+1)\cdot(2K+1)$, probably via the Damgard-Landrock Theorem. This approach converts Rabin's algorithm form a $O(\log^3 N)$ test to a $O(\log^4 N)$, but we believe that a general-purpose system needs the additional security.

It must be emphasised that we have not produced a guaranteed $O(\log^4 N)$ primality test: merely one that we do not believe we can break by the technology we know. It would be tempting to conjecture that, with an appropriate constant of proportionality, this test is guaranteed never to return "$N$ is probalby prime" when in fact it is composite. The closest result to this we know of is a statement by Lenstra [**?**] (see also Koblitz [Kobl87]) that, if suitable assumptions similar to the generalised Riemann hypothesis are made, the $70\log^2 N$ values suffice, which would be a $O(\log^5 N)$ primality test.

```
          The Pomerance et al. Modifications

[ 0a] PomeranceList:=[25326001::I,161304001::I,960946321::I,1157839381::I,
[ 0b]                  -- 3215031751::I, -- has a factor of 151
[ 0c]                 3697278427::I, 5764643587::I, 6770862367::I,
[ 0d]                 14386156093::I, 15579919981::I, 18459366157::I,
[ 0e]                 18459366157::I, 21276028621::I]::(List I)
[ 0f] PomeranceLimit:=(25*10**9)::I
[ 1] prime? n ==
[ 2]    n < two => false
[ 3]    n < nextSmallPrime => member?(n, smallPrimes)
[ 4]    not one? gcd(n, productSmallPrimes) => false
[ 5]    n < nextSmallPrimeSquared => true
[ 6]    nm1:=n-1
[ 7]    q := (nm1) quo two
[ 8]    for k in 1.. while not odd? q repeat q := q quo two
[ 9]    -- q = (n-1) quo 2**k for largest possible k
[10]    mn := minIndex smallPrimes
[10a]   n < PomeranceLimit =>
[10b]     rabinProvesCompositeSmall(2::I,n,nm1,q,k) => return false
[10c]     rabinProvesCompositeSmall(3::I,n,nm1,q,k) => return false
[10d]     rabinProvesCompositeSmall(5::I,n,nm1,q,k) => return false
[10e]     member?(n,PomeranceList) => return false
[10f]     true
[10g]   rootsMinus1 := [] -- the empty set
[10h]   count2Order := new(k,0) -- vector of k zeros
[11]    for i in mn+1..mn+10 repeat
[12]      rabinProvesComposit(smallPrimes i,n,nm1,q,k) => return false
[12a]     import IntegerRoots(I)
[12b]     q > 1 and perfectSquare?(3*n+1) => false
[12c]     ((n9:=n rem (9::I))=1 or n9 = -1) and perfectSquare(8*n+1) => false
[12d]     -- Both previous tests fro Damgard & Landrock
[12e]     currPrime:=smallPrimes(mn+10)
```

```
[12f]     while count2Order(k) = 0 repeat
[12g]           currPrime := nextPrime currPrime
[12h]           rabinProvesComposite(currPrime,n,nm1,q,k) => false
[13]    true
[14]
[15] rabinProvesComposite(p,n,mn1,q,k) ==
[16]    -- nm1 = n-1 = q*2**k; q odd
[17]    -- probability false for n composite is < 1/4
[18]    -- for most n this probability is much less than 1/4
[19]    t := powmod(p, q, n)
[19a]   if t=mn1 then count2Order(1):=count2Order(1)+1
[20]    -- neither of these cases tells us anything
[21]    if not (one? t or t = nm1) then
[22]       for j in 1..k-1 repeat
[22a]          oldt := t
[23]          t := mulmod(t, t, n)
[24]          one? t => return true
[25]          -- we have squared somethiong not -1 and got 1
[26]          t = mn1 =>
[26a]              rootsMinus1:=union(rootsMinus1,oldt)
[26b]              # rootsMinus1 > 2 => return true
[26c]              count2Order(j+1):=count2Order(j+1)+1
[27]              leave
[28]       not (t = nm1) => return true
[29]    false
```

O(Log^4 N) Modifications

```
[ 0a] PomeranceList:=[25326001::I,161304001::I,960946321::I,1157839381::I,
[ 0b]                  -- 3215031751::I, -- has a factor of 151
[ 0c]                 3697278427::I, 5764643587::I, 6770862367::I,
[ 0d]                 14386156093::I, 15579919981::I, 18459366157::I,
[ 0e]                 18459366157::I, 21276028621::I]::(List I)
[ 0f] PomeranceLimit:=(25*10**9)::I
[ 1] prime? n ==
[ 2]    n < two => false
[ 3]    n < nextSmallPrime => member?(n, smallPrimes)
[ 4]    not one? gcd(n, productSmallPrimes) => false
[ 5]    n < nextSmallPrimeSquared => true
[ 6]    nm1:=n-1
[ 7]    q := (nm1) quo two
[ 8]    for k in 1.. while not odd? q repeat q := q quo two
[ 9]    -- q = (n-1) quo 2**k for largest possible k
[10]    mn := minIndex smallPrimes
[10a]   n < PomeranceLimit =>
[10b]     rabinProvesCompositeSmall(2::I,n,nm1,q,k) => return false
[10c]     rabinProvesCompositeSmall(3::I,n,nm1,q,k) => return false
[10d]     rabinProvesCompositeSmall(5::I,n,nm1,q,k) => return false
[10e]     member?(n,PomeranceList) => return false
[10f]     true
[10g]   rootsMinus1 := [] -- the empty set
[10h]   count2Order := new(k,0) -- vector of k zeros
[11]    for i in mn+1..mn+10 repeat
[12]      rabinProvesComposit(smallPrimes i,n,nm1,q,k) => return false
```

```
[12a]     import IntegerRoots(I)
[12b]     q > 1 and perfectSquare?(3*n+1) => false
[12c]     ((n9:=n rem (9::I))=1 or n9 = -1) and perfectSquare(8*n+1) => false
[12d]     -- Both previous tests fro Damgard & Landrock
[12e]     currPrime:=smallPrimes(mn+10)
[12f]     probablySav=fe:=tenPowerTwenty
[12g]     while count2Order(k) = 0 or n > probablySaferepeat
[12h]           currPrime := nextPrime currPrime
[12i]           probablySafe:=probablySafe*(100::I)
[12j]           rabinProvesComposite(currPrime,n,nm1,q,k) => false
[13]    true
[14]
[15] rabinProvesComposite(p,n,mn1,q,k) ==
[16]    -- nm1 = n-1 = q*2**k; q odd
[17]    -- probability false for n composite is < 1/4
[18]    -- for most n this probability is much less than 1/4
[19]    t := powmod(p, q, n)
[19a]   if t=mn1 then count2Order(1):=count2Order(1)+1
[20]    -- neither of these cases tells us anything
[21]    if not (one? t or t = nm1) then
[22]       for j in 1..k-1 repeat
[22a]          oldt := t
[23]           t := mulmod(t, t, n)
[24]           one? t => return true
[25]           -- we have squared somethiong not -1 and got 1
[26]           t = mn1 =>
[26a]              rootsMinus1:=union(rootsMinus1,oldt)
[26b]              # rootsMinus1 > 2 => return true
[26c]              count2Order(j+1):=count2Order(j+1)+1
[27]              leave
[28]       not (t = nm1) => return true
[29]    false
```

# Chapter 13

# Finite Fields in Axiom (Grabmeier/Scheerhorn)

This was written by Johannes Grabmeier and Alfred Scheerhorn. Matthew Salomone's video course[Salo16] provides useful background material.

Finite fields play an important role in mathematics and in many applications as coding theory or factorizing polynomials in computer algebra systems. They are the finite sets which have a computational structure as the classical fields of rational or complex numbers, i.e. addition + and multiplication with inverses and the usual group axioms as commutativity and associativity laws and their interaction via the distributivity laws. For further details see any book on algebra or our preferred reference for finite fields [Lidl83].

The finite fields are classified: For each prime power $q = p^n$ there is up to isomorphism exactly one finite field of these size and there are no more. So far this looks nice, easy and complete. However, there are different constructions of a finite field of a given size $q$, each having different advantages and disadvantages. This paper deals with such constructions and implementations in the computer algebra system Axiom, various isomorphims and embeddings. We have three different kinds of constructions, namely polynomial basis representation, normal basis representation, and cyclic group representation.

The various advantages and disadvantages which will be discussed along with the special implementations of the representations in the respective sections. All are strongly connected with the construction of irreducible polynomials which have additional properties. The user of Axiom may choose the representation which best meets the needs for his applications. For each type we have provided automatic choices as well as the liberty to use a favourite polynomial. In addition there are implementations for mechanisms to convert the data from one representation to the other.

The paper is organized as follows: For convenience of the readers we first recall some basic facts from the theory of finite fields. Then we introduce our category design of the finite field world in Axiom. Next comes the description of all the functions which are valid and useful for every finite field. We employ the abstract datatype concepts of Axiom, which allows to implement such functions in the default packages of these categories. This makes an implementation in the different domains superfluous. Using a special kind of representation the implementation of many functions can be improved in order to have a more effcient computation. We describe these improved, additional domain implementations in the sections

according to the special representations. Section 13.9 is devoted to the various constructions of polynomials. In section 13.11 we finally give results of time comparison of the various representations.

## 13.1 Basic theory and notations

Salomone's lectures provide background material for this section.

We denote a finite field with $q = p^r$ elements, $p$ a prime and $r \in \mathbb{N}$, by $GF(q)$. The *prime field GF(p)* can be constructed as $\mathbb{Z}/p\mathbb{Z} = \{0, 1, \ldots, p-1\}$. The finite field $GF(q)$ is an algebraic extension of the field $GF(p)$ and isomorphic to the splitting field of $X^q - X$ over $GF(p)$. Let $\alpha, \beta \in GF(q)$ and $c \in GF(p)$, then we have $(\alpha+\beta)^p = \alpha^p + \beta^p$ and $c\alpha^p = (c\alpha)^p$. Therefore powering with $p$ or powers of $p$ is a linear operation over $GF(p)$. Let $E = GF(q^n)$ be an extension of $F = GF(q)$ of degree $n \in \mathbb{N}$. The automorphism group of $E$ over $F$ is cyclic of order $n$ and generated by

$$\sigma : \alpha \to \alpha^q$$

which is called a *Frobenius automorphism*.

Let $f \in F[X]$ be a monic, irreducible polynomial of degree $n$. Then

$$E \simeq F[X]/(f)$$

where $(f) = f \cdot F[X]$ denotes the principal ideal generated by $f$, and the isomorphism is given by $\alpha \mapsto (X \bmod f)$, where $\alpha$ is a root of $f$ in $E$. $\alpha$ generates a *polynomial basis* $\{1, \alpha, \ldots, \alpha^{n-1}\}$ of $E$ over $F$. Every element $\beta \in E$ can be expressed uniquely in the form

$$\beta = \sum_{i=0}^{n-1} b_i \alpha^i$$

This kinds of representing elements of $E$ is called *polynomial basis representation*.

Let $\alpha \in E$, the monic, irreducible polynomial $f \in F[X]$ with $f(\alpha) = 0$ is called the *minimal polynomial* of $\alpha$ over $F$. All the roots of $f$ are given by the set of *conjugates*

$$\{\alpha, \alpha^q, \ldots, \alpha^{q^n}\}$$

of $\alpha$ of $F$. Therefore

$$f = \prod_{i=0}^{n-1} (X - \alpha^{q^i})$$

The *trace* $T_{E/F}(\alpha)$ and the *norm* $N_{E/F}(\alpha)$ of $\alpha$ over $F$ are defined by the sum and the product of the conjugates of $\alpha$, respectively,

$$T_{E/F}(\alpha) = \sum_{i=0}^{n-1} \alpha^{q^i}, \quad N_{E/F}(\alpha) = \prod_{i=0}^{n-1} \alpha^{q^i} = \alpha^{\frac{q^n-1}{q-1}} \tag{1}$$

These values can be read off from the minimal polynomial

$$f = \sum_{i=0}^{n} f_i X^i \in F[X]$$

of $\alpha$ over $F$:

$$T_{E/F}(\alpha) = -f_{n-1}, \quad N_{E/F}(\alpha) = (-1)^n f_0$$

The *degree* of $\alpha$ over $F$ is the degree of the smallest subfield of $E$ over $F$, which contains $\alpha$, i.e. the minimal integer $d > 0$ for which

$$\alpha^{q^d} = \alpha$$

If $\alpha$ has degree $d$ over $F$, the minimal polynomial $m_\alpha(X)$ of $\alpha$ then has degree $d$, too.

The multiplicative group $E^*$ of $E$ is cyclic of order $q^n - 1$. A generator of this group is called a *primitive element* of $E$ and the minimal polynomial of such an element is called a *primitive polynomial*. Every nonzero element $\beta \in E$ can be expressed as a power of $\alpha$:

$$\beta = \alpha^e$$

where $0 \leq e < q^n - 1$ is uniquely determined. This kind of representing the elements of $E$ is called *cyclic group representation* of $E$. The exponent $e$ is called the *discrete logarithm* of $\beta$ to base $\alpha$ denoted by $e = log_\alpha(\beta)$. Note, that exponentiaion

$$\mathbb{Z} \times E \to E : (e, \alpha) \mapsto \alpha^e$$

defined a $\mathbb{Z}$-module structure on the multiplicative group $E^*$.

Analogically one can define a module structure on the additive group of $E$ in the following way.

Let $\circ : F[X] \times E \to E$ be defined by

$$\sum_i a_i X^i \circ \alpha := \sum_i a_i \alpha^{q^i}$$

Then we get for $\alpha \in E$, $g = \sum_i g_i X^i$, $f = \sum_j f_j X^j \in F[X]$,

$$g \circ (f \circ \alpha) = g \circ (\sum_j f_j \alpha^{q^j}) = \sum_i g_i (\sum_j f_j \alpha^{q^j})^{q^i} = \sum_{i,j} g_i f_j \alpha^{q^{i+j}} = (g \cdot f) \circ \alpha$$

This proves that the operation $\circ$ defines an $F[X]$-module structure on the additive group of $E$.

For $\alpha \in E$ the annihilator ideal $Ann_\alpha = \{f \in F[X] : f \circ \alpha = 0\}$ of $\alpha$ is generated by a single polynomial of $F[X]$, since $F[X]$ is a principle ideal domain. We call the unique, monic generator of $Ann_\alpha$ the *linear associated order* of $\alpha$ over $F$, denoted by $\mathrm{Ord}_q(\alpha)$:

$$\{f \in F[X] : f \circ \alpha = 0\} = \mathrm{Ord}_q(\alpha) F[X]$$

Since $(X^n - 1) \circ \alpha = \alpha^{q^n} - \alpha = 0$ for all $\alpha \in E$, $\mathrm{Ord}_q(\alpha)$ divides $(X^n - 1)$.

If $\mathrm{Ord}_q(\alpha) = (X^n - 1)$ then there exists no polynomial of degree less $n$ in F[X] annihilating $\alpha$, i.e. if $f \in F[X]$ is of degree $\deg(f) < n$ and $f \circ \alpha = 0$, then $f = 0$. That is

$$\sum_{i=0}^{n-1} f_i \alpha^{q^i} = 0 \text{ implicates } f_i = 0, 0 \leq i < n$$

Therefore $\{\alpha, \alpha^q, \ldots, \alpha^{a^{n-1}}\}$ constitutes a basis of $E$ over $F$, called a *normal basis*. Therefore, we call an element with linear associated order $(X^n - 1)$ *normal (over F)*. In this case every element $\beta \in E$ can be expressed uniquely in the form

$$\beta = \sum_{i=0}^{n-1} b_i \alpha^{q^i} = b(X) \circ \alpha, \quad b(X) = \sum_{i=0}^{n-1} b_i X^i \in F[X]$$

This kind of representing elements of $E$ is called *normal basis representation* of $E$. The polynomial $b(X)$ of degree $\deg(b) < n$ is called the *linear associated logarithm* of $\beta$ to base $\alpha$ over $F$, denoted by $b(X) = \mathrm{Log}_{q,\alpha}(\beta)$. It is uniquely determined modulo $\mathrm{Ord}_q(\alpha)$.

Again, with $\alpha$ all of its conjugates are normal over $F$, too. Therefore we call an irreducible polynomial $f \in F[X]$ of degree $n$ normal over $F$, if the roots of $f$ in $GF(q^n)$ are linearly independent.

## 13.2 Categories for finite field domains

Each domain in Axiom which represents a finite field is an object in the category

`FFIELDC FiniteFieldCategory`

This category defines functions which are typical for finite sets as `order` or `random` and those which are typical for finite fields and do not depend on the ground field. An example is `primeFrobenius`, which implements the Frobenius automorphism with respect to the prime field, or functions concerning the cyclic multiplicative group structure as `primitiveElement`, see section 13.3.3

Functions which need to know the *groundfield* $F$ are defined in

`FAXF(F)  FiniteAlgebraicExtensionField(F)`

These are functions considering the extension field as an algebra of finite rank over $F$, see section 13.3.1, and functions concerning the $F[X]$-module structure of the additive group of the extension field, see section 13.3.2

Every finite field can be considered as an extension field of each of its subfields, in particular the prime field. In Axiom every finite field constructor is implemented to belong to the category `FAXF(F)` for exactly one $F$. Constructors as `FF` are considered as extensions of its prime field, while others use the explicit given ground field. Note that even a prime field is an extension of itself. Mainly for technical reasons, but also for restrictions of the present compiler, every finite field datatype in Axiom is an extension of exactly *one* subfield, usually called the *ground field*. Otherwise functions like `extensionDegree` would depend on the various ground fields and this would be not very convenient. For possible enhancements in future releases, see section **??**. NOt that this of course does not affect the ability to build arbitrary towers of finite field extensions.

## 13.3 General finite field functions

Let $E = GF(q^n)$ and $F = GF(q)$ be represented by `E` of type `FAXF(F)` and `F` respectively. The characteristic of $E$ is given by `characteristic()$E`.

### 13.3.1 $E$ as an algebra of rank $n$ over $F$

The degree $n$ of $E$ over $F$ is returned by `extensionDegree()$E`

The `definingPolynomial()$E` yields the polynomial $f \in F[X]$ by which the field extension $E$ over $F$ is defined. The element of `E` representing $(X \bmod f(X))$, i.e. a root of the defining polynomial, is returned by `generator()$E`.

A basis of $E$ over $F$ is yielded by calling `basis()$E`. In the polynomial basis representation and cyclic group representation this is the polynomial basis generated by `generator()$E`. In the normal basis representation it is the normal basis generated by this element.

Let $\alpha = \sum_{i=0}^{n-1} a_i \alpha_i$, where $a_i$ are elements of `F` and $(\alpha_0, \ldots, \alpha_{n-1})$ =`basis()$E`. The `coordinates`$(\alpha)$ function computes the coordinate vector $(a_0, \ldots, a_{n-1})$ of type `Vector F` of $\alpha$ over $F$. The `represents` function is the inverse of `coordinates` and yields $\alpha$, if applied to the vector $(a_0, \ldots, a_{n-1})$.

The implementation of these functions is straightforward in the polynomial and normal basis representation. For the cyclic group representation see section 13.6.3

The functions `degree`, `trace`, and `norm` applied to $\alpha \in E$ compute the degree, trace, and norm of $\alpha$ over $F$, respectively. For their implementation see the sections according to the special representations.

There are two ways of computing the minimal polynomial $m_\alpha(X)$ of $\alpha \in E$ over $F$. The first method is to compute

$$m_\alpha(X) = \prod_{i=0}^{d-1} (X - \alpha^{q^d})$$

where $d$ denotes the degree of $\alpha$ over $F$. It unfortunately needs $d(d-1)/2$ multiplications in $E$ and $(d-1)$ exponentiations by $q$.

The second method is to compute first a matrix $M \in F^{n \times (d+1)}$ whose $i$-th column is the coordinate vector of $\alpha^i$ w.r.t. an arbitrary base of $E$ over $F$, for $0 \leq i \leq d$. Then there exists a vector $b = (b_0, \ldots, b_d)^t$ in the nullspace of $M$, which can be computed. Hence, the polynomial

$$m_\alpha = X^d + \frac{1}{b^d} \sum_{i=0}^{d-1} b_i X^i$$

has the root $\alpha$ and is of correct degree, hence it is the minimal polynomial. This method requires only about $(d-1)$ multiplications in $E$ and $O(d^2 n)$ operations in $F$. Which approach is more time efficient depends on the relation between the computation time of operations in $F$ and multiplications in $E$. Since multiplication in $E$ is cheap in a cyclic group represention of $E$ (see section 13.6.1) we use the first method there. The second method was found to be more time efficient for the other representations.

Let $d$ be a divisor of $n$ and $L$ be a subfield of $E$ of degree $d$ over $F$. For the functions `trace`, `norm`, `basis`, and `minimalPolynomial` we offer a second version, which gets $d$ as an additional parameter. `basis`$(d)$ returns a basis of $L$ over $F$. In the polynomial basis representation and cyclic group representation, this is the polynomial basis generated by `norm(primitiveElement()$E,d)`. In the normal basis representation it is the normal basis generated by `trace(generator()$E,d)`. For $\alpha \in E$ `trace`$(\alpha, d)$ and `norm`$(\alpha, d)$ compute

$T_{E/L}(\alpha)$ and $N_{E/L}(\alpha)$, respectively, by

$$T_{E/L}(\alpha) = \sum_{i=0}^{n/d-1} \alpha^{q^{id}} \quad \text{and} \quad N_{E/L}(\alpha) = \prod_{i=0}^{n/d-1} \alpha^{q^{id}}$$

Similarly we get the minimal polynomial of $\alpha$ over $L$ by

$$\prod_{i=0}^{m-1} (X - \alpha^{q^{id}})$$

where $m$ denotes the degree of $\alpha$ over $L$.

### 13.3.2 The $F[X]$-module structure of $E$

In this section we discuss the three operations `linearAssociatedExp`, `linearAssociatedLog`, and `linearAssociatedOrder`.

For $f = \sum_{i=0}^{n-1} f_i X^i \in F[X]$ and $\alpha \in E$ the function `linearAssociatedExp`$(\alpha, f)$ computes $f \circ \alpha$ in a straightforward way via

$$f \circ \alpha = \sum_{i=0}^{n-1} f_i \alpha^{q^i}$$

For $\alpha \in E$ we want to construct the monic polynomial $\texttt{Ord}_q(\alpha)$ of least degree $d$ such that

$$\mathrm{Ord}_q(\alpha) \circ \alpha = 0$$

This suggests to analyze the linear relations between the $q$-powers of $\alpha$. Therefore we define $m \in F^{n \times n}$ to be the matrix whose $i$-th column is the coordinate vector of $\alpha^{q^i}$ w.r.t. a basis $B$ of $E$ over $F$, for $0 \leq i \leq n$. Its rank equals the degree $d$ of $\texttt{Ord}_q(\alpha)$. If $m$ has rank $n$ then $\alpha$ is normal in $E$ over $F$ and $\texttt{Ord}_q(\alpha) = X^n - 1$. Otherwise a vector $b = (b_0, \ldots, b_{n-1})^t$ of the nullspace of $M$ is computed. It is easy to see that we can choose $b_{d+1}, b_{d+2}, \ldots, b_{n-1} = 0$ and finally we get

$$\mathrm{Ord}_q(\alpha) = X^d + \frac{1}{b^d} \sum_{i=0}^{d-1} b_i X^i$$

which is computed by `linearAssociatedOrder`$(\alpha)$ in Axiom.

For $\beta$ and $\alpha \in E$ `linearAllocatedLog`$(\alpha, \beta)$ computed $\texttt{Log}_{q,\alpha}(\beta)$, if it exists, i.e. the unique polynomial $f = \sum_i f_i X^i \in F[X]$ of degree less then $\deg(\mathrm{Ord}_q(\alpha))$, for which $f \circ \alpha = \beta$.

As before we solve a system of linear equations

$$M \cdot (f_0, f_1, \ldots, f_{n-1})^t = b^t$$

where $M$ is as above and $b$ denotes the coordinate vector of $\beta$ w.r.t. the basis $B$. If there exists a solution satisfying $f_d, f_{d+1}, \ldots, f_{n-1} = 0$ for the degree $d$ of $\mathrm{Ord}_q(\alpha)$, then

$$\mathrm{Log}_{q,\alpha}(\beta) = \sum_{i=0}^{d-1} f_i X^i$$

otherwise `linearAssociatedLog`$(\alpha, \beta)$ returned `"failed"`.

The function `linearAssociatedLog`$(\alpha)$ with only one argument computed the linear associated logarithm of $\alpha$ to the base given by `normalElement()$E`.

### 13.3.3 The cyclic group $E^*$

The inheritance mechanisms of Axiom provide for each multiplicative structure automatically an implementation of the exponentiation by the defaulting repeating squaring algorithm. Finite Fields are an example where it makes sense to overwrite this defaulting function. In all the implementations we first reduce the exponent modulo $q^n - 1$ and then do repeated squaring or other algorithms.

To compute the multiplicative order $e := \text{order}(\alpha)$ of $\alpha \in E$ we proceed as follows: We start with $e := q^n - 1$. For every prime $p$ of $(q^n - 1)$ we divide $e$ by $p$: $e := e/p$, as long as $e$ is divisible by $p$ and $\alpha^{e/p} = 1$.

Note that this implementation requires factorizing the order $(q^n - 1)$ of the multiplicative group. Other functions as `discreteLog` also need these factors. Since factoring $(q^n - 1)$ is time expensive, the factorization of $(q^n - 1)$ is stored in a global variable

```
facOfGroupSize: List Record(factor:Integer,exponent:Integer)
```

in the respective domain. To make this information available to the category for default implementations, we export the function

```
factorsOfCyclicGroupSize()
```

which returns the factorization. $(q^n - 1)$ is factored only the first time this function is called.

### 13.3.4 Discrete logarithm

For the computation of discrete logarithms we implemented the *Silver-Pohlig-Hellman algorithm* combined with the *baby steps-giant steps* technique of Shank. A nice survey of discrete logarithm algorithms included the algorithms we used is given by Odlyzko in [Odly85].

The Silver-Pohlig-Hellman algorithm breaks the problem of computing discrete logarithms in a cyclic group of order $q^n - 1$ down to computing discrete logarithms in cyclic groups of order $p$, where $p$ ranges over all prime factors of $q^n - 1$. For a detailed description see [Odly85] or [Pohl78].

To compute discrete logarithms in cyclic groups of prime order $p$ we use Shank's algorithm implemented by

```
shanksDiscLogAlgorithm:(M,M,NonNegativeInteger) ->
                        Union(NonNegativeInteger,"failed")
```

in the package `DiscreteLogarithmPackage(M)` and `M` has to be of type `Join(Monoid,Finite)`, i.e. a finite multiplicative Monoid.

```
shanksDiscLogAlgorithm(b,a,p)
```

computes $e$ with $b^e = a$ assuming that $b$ and $a$ are elements of a finite cyclic group of order $p$. If no such $e$ exists, it returns `"failed"`.

Here is a brief description of the algorithm: Let $\tilde{p}$ be an integer close to $\sqrt{p}$. First we create a key-access table with entries $(b^k, k)$. Then we look whether the table contains an entry with key $a \cdot b^{-j\tilde{p}}$ and get $k$ with $a \cdot b^{-j\tilde{p}} = b^k$ for the smallest $j = 0, 1, \ldots, \lceil p/\tilde{p} \rceil - 1$ or `"failed"`. In the first case the result is $e = k + j\tilde{p}$.

In the Silver-Pohlig-Hellman algorithm for a given base $\beta \in E$, the first argument $b$ when calling `shanksDiscLogAlgorithm(b,.,.)` is for a fixed prime factor $p$ of $(q^n - 1)$ always the same. To compute logarithms to the base given by `primitiveElement()$E` efficiently,

the tables needed by Shanks algorithm are precomputed and stored in the global variable

```
discLogTable : Table(PI,Table(PI,NNI))
```

in `E`. It is initialized at the first call of the function `discreteLog`. Here `PI` abbreviates `PositiveInteger` and `NNI` abbreviates `NonNegativeInteger`.

To implement the discrete logarithm function on category level in `FiniteFieldCategory`, we have to make this data available to the category. This is done by exporting the function

```
tableForDiscreteLogarithm: Integer -> Table(PI,NNI)
```

Called with a prime divisor $p$ of $q^n-1$ as argument, it returns a table of size roughly $\tilde{p} \approx \sqrt{p}$, whose $k$-th entry for $0 \leq k < \tilde{p}$ is of the form

```
[lookup(a**k),k) : Record(key:PI,entry:PI)
```

where $a = \alpha^{(q^n-1)/p}$

We implemented two functions for discrete logarithms:

```
discreteLog: E     -> NonNegativeInteger
discreteLog: (E,E) -> Unioin(NonNegativeInteger,"failed")
```

The first one computes discrete logarithms to the base `primitiveElement()$E` using the precomputed tables. `discreteLog(b,a)` computes $\log_b(a)$ if $a$ belongs to the cyclic group generated by $b$ and fails otherwise. This function does not use the table `discLogTable`. No initialization of this table is performed using the second function.

### 13.3.5 Elements of maximal order

The functions

```
primitiveElement()$E
normalElement()$E
```

yields a primitive element of $E$ and a generator of a normal basis of $E$ over $F$, respectively. The first having maximal multiplicative order $(q^n-1)$, the second maximal linear associated order $(X^n-1)$.

To compute elements of maximal order there exist algorithms which construct elements of high order from elements of low order. For a unifying module theoretic approach see Lüneburg[Lune87] chap.IV. For the construction of primitive elements see [Rybo89] where an algorithm is described which is originally given in [Vars81]. Algorithms for finding generators of normal basis are described in [Gath90], [Lens91], or [Pinc89].

Experiments have shown that in practice it is more efficient to simply run through the field and test until an element of maximal order is found. However, not very many theoretical results are known on deterministic search procedures. In some situations there are results depending on the Extended Riemann Hypothesis, see [Shou92].

To aviod searching a second time for the same element, we store the results after the first computation of such an element using the helper functions

```
createPrimitiveElement()$E
createNormalElement()$E
```

The functions

```
primitive?
normal?
```

check whether the multiplicative resp. linear associated order of an element is maximal.

The primitivity of an $\alpha$ in $E$ is tested using the fact that $\alpha$ is primitive in $E$ if and only if for all prime factors $p$ of $(q^n - 1)$ holds:

$$\alpha^{(q^n-1)/p} \neq 1$$

see e.g. Theorem 2 of chap. I.X.1.3 in [Lips81].

For testing whether a given element $\alpha$ generates a normal basis of $E$ over $F$ we use Theorem 2.39 of [Lidl83]: $\alpha \in E$ generates a normal basis of $E$ over $F$ if and only if $(X^n - 1)$ and $\sum_{i=0}^{n-1} \alpha^{q^i} X^{n-1-i}$ are relatively prime in $E[X]$

### 13.3.6 Enumeration of elements of $E$

The number of elements of $E$ can be found out by calling `size()$E`. The elements of `E` are enumerated by

```
index: PositiveInteger -> E
```

with inverse

```
lookup: E -> PositiveInteger
```

These functions implement a bijection between `E` and the set of positive integers $\{1, 2, \ldots, q^n\}$. They allow iterating over all field elements. `lookup` can be used to store field elements in a variable of type `PositiveInteger`. This is often less memory expensive than storing the field element which may be represented in a complicated way. For time efficiency reasons these functions have different implementations according to the different representations. ALl of them have in common that `index`$(q^n) = 0\$E$ and `lookup`$(0\$E) = q^n$.

### 13.3.7 Conversion between elements of the field and its groundfield

To check whether a given element $\alpha$, representation by `a` in the field `E`, belongs to its groundfield $F$ use `inGroundField?(a)`. If $\alpha$ belongs of $F$, datatype conversion is provided by `retract(a)`.

Embedding from `F` to `E` is done using `coerce` abbreviated by `::` in Axiom. If `a` is the representation of $\alpha$ in `F`, then `(a::E)` is the element of `E` representing $\alpha$.

All these functions depend on the representation used and are explained in the sections according to the special representations.

## 13.4 Prime field

Let $p$ be a prime number. Since $GF(p) \simeq \mathbb{Z}/p\mathbb{Z}$ in Axiom the internal representation of elements of the domains

```
IPF(p)    InnerPrimeField(p)
PF(p)     PrimeField(p)
```

is `IntegerMod(p)`, from which most functions are inherited. The only difference between `IPF` and `PF` is, that in `PF` it is checked whether the parameter $p$ is prime while this is not checked in `IPF`.

Many functions as `trace` or `inGroundField?` are trivial for a prime field. For a human being there is no problem to consider a prime field as an extension of degree 1 of itself. For recursions depending e.g. on the extension degree and simply for completeness, we have decided to make `PrimeField` an `FiniteAlgebraicExtensionField` of itself. The trivial implementations include

```
normalElement()    == 1
inGroundField?(a)  == true
generator()        == 1
```

Since the value returned by `generator()` should be a root of the defining polynomial of the field extension, we had to code `definingPolynomial` to be $X - 1$ and not e.g. $X$.

### 13.4.1 Extension Constructors of Finite Fields

There are three choices to make when one wants to construct a finite field as an extension of a ground field in Axiom.

The first choice is the type of representation. It can be remainder classes of polynomials, see section 13.5, exponents of a primitive element, see section 13.6, or a normal basis representation, see section 13.7. The part of the abbreviation of the corresponding domain constructors are `FF`, `FFCG`, and `FFNB`, respectively.

Secondly, we have to decide which ground field we choose, either the prime field or any other subfield. In the first case the first parameter of all domain constructors is just a prime number and one has to use the names above. In the other case the first parameter is the domain constructed recursively to represent the chosen subfield.

The second parameter governs the extension. All constructions depend on an irreducible polynomial, whose degree is the extension degree and has, if necessary, additional properties. If one doesn't care about this polynomial, one has to give the degree as the second parameter and the polynomial will be chosen approximately by Axiom. In the case where we define the prime field by supplying a prime number, this is the only choice. In the other case one has to append the letter `X` to the receive the domain constructors `FFX`, `FFCGX`, and `FFNBX`, respectively. If one wants to supply one's favorite polynomial as the second parameter we have to substitute the letter `X` by `P`.

Here are a few datatype constructions for these nine possibilities: `FF(2,10)` implements an extension of the prime field `PF 2` of degree 10. Axiom chooses an irreducible polynomial of degree 10 for this polynomial basis representation.

`FFNBX(FFCGP(3,f),5)` implements an extension of the field with $3^n$ elements, represented as exponents of a primitive element, where $f$ is a primitive polynomial of degree $n$. The extension of degree 5 is realized by Axiom by choosing a normal polynomial of degree 5 with coefficients in `FFCGP(3,f)`.

As overloading of constructor names is not supported by the current compiler, we had to create all these different names as explained above. As soon as the new compiler will support this we may consider to unify these domain domain names, see section **??**.

## 13.5 Polynomial basis representation

Let $E = GF(q^n)$ be an extension of degree $n$ over $F = GF(q)$. Then

$$E \simeq F[X]/(f)$$

where $f \in F[X]$ is an arbitrary monic irreducible polynomial of degree $n$. If $\alpha$ is a root of $f$ in $E$, then $\{1, \alpha, \ldots, \alpha^{n-1}\}$ constitutes a basis of $E$ over $F$ and we can write all elements $\beta \in E$ uniquely in the form:

$$\beta = \sum_{i=0}^{n-1} b_i \alpha^i, \quad b_i \in F$$

This kind of representation is used in the domains

```
FFP   FiniteFieldExtensionByPolynomial
FFX   FiniteFieldExtension
IFF   InnerFiniteField
FF    FiniteField
```

The only difference between these domains are the different natures of their parameterization, see section 13.4.1. The datatype `InnerFiniteField` extends the prime field `InnerPrimeField`, see section 13.4. Let `F` be the domain representing $F$ and `E:=FFP(F,f)` the extension of $F$ defined by $f$.

For the internal representation of elements of $E$ we use polynomials modulo $f$. This structure is in Axiom implemented by the domain

```
SAE(F,SUP(F),f)  SimpleAlgebraicExtension(F,SUP(F),f)
```

Here `SUP(F)` abbreviates `SparseUnivariatePolynomial(F)` a domain representing polynomials over $F$. Most arithmetic operations are inherited from this domain. There are only a few functions which have a special implementation.

The imbedding of `F` in `E` is obvious: $\sum_{i=0}^{n-1} a_i \alpha^i$ is in $F$ if and only if $a_1, a_2, \ldots, a_{n-1} = 0$. Using this, the functions

```
retract:       E -> F
coerce:        F -> E
inGroundfield? E -> Boolean
```

are implemented in the obvious way.

To check whether $\beta$ is normal in $E$ over $F$ we proceed as follows. Let $M \in F^{n \times n}$ be the matrix, whose $i$-th column is the coordinate vector of $\beta^{q^i}$ with respect to the polynomial basis for $0 \leq i < n$. The element $\beta$ is normal if and only if the rank of $M$ equals $n$.

The coordinates of $\beta\alpha^i$ collected in a matrix similar as before give the *regular matrix representation* (see e.g. chap.7.3 in [Jaco85]) of $\beta$. The functions `trace` and `norm` compute the trace and norm of $\beta$ over $F$ by computing the trace and determinant of this matrix, respectively.

## 13.6 Cyclic group representation

In this section we make use of the fact, that the multiplicative group of a finite Field $E$ with $q^n$ Elements is cyclic of order $q^n - 1$. Therefore it is isomorphic to $\mathbb{Z}/(q^n - 1)\mathbb{Z}$. Once a primitive element $\alpha$ of $E$ is fixed (i.e. a generator of the multiplicative group), ever nonzero

element $\beta$ of $E$ is uniquely determined by its discrete logarithm $e$ to $\alpha$, i.e. the element $0 \leq e \leq q^n - 1$ with $\alpha^e = \beta$.

In the three domains

```
FFCGP   FiniteFieldCyclicGroupExtensionByPolynomial
FFCGX   FiniteFieldCyclicGroupExtension
FFCG    FiniteFieldCyclicGroup
```

the nonzero field elements are represented by powers of a fixed primitive element. Let `F` be a finite field domain representing $F$, $E$ the extension of $F$ defined by the monic, irreducible polynomial $f(x) \in F[X]$ with root $\alpha \in E$. Let $\alpha$ be primitive in $E$ and `E:=FFCGP(F,f)` be the domain representing $E$.

As the fixed primitive element used for the representation, we take the root $\alpha$ of $f$. It is returned by calling `generator()$E`, which is equal to the function `primitiveElement()$E` in this representation.

The aim of a cyclic group representation of finite fields is to offer a very fast field arithmetic. All operation concerning the multiplicative structure of the field are quite easy to compute. To have a quick addition one has to store a Zech (Jacobi) logarithm table in memory, see setion 7.2. This table is of size about $q^n/2$. For efficiency reasons we also want to use `SmallInteger` and not `Integer` as the internal representation of the field elements. Therefore we restricted ourself to a field size of maximal $2^{20}$ elements:

```
if sizeFF > 2**20 then
  error "field too large for cyclic group representation"
```

A nonzero element $\beta \in E$ is represented by the unique $n \in \{0, 1, \ldots, q^n - 2\}$ of type `SmallInteger` with $\alpha^n = \beta$ and $0 \in E$ is represented by the `SmallInteger` $-1$.

### 13.6.1 Operations of multiplicative nature

The implementation of the operations concerning the multiplicative group is very easy. Since $\alpha^n \cdot \alpha^m = \alpha^{n+m}$, multiplication of nonzero elements becomes a `SmallInteger` addition modulo $q^n - 1$. Similarly the exponentiation of field elements and the norm function are done by a modular `SmallInteger` multiplication. Inversion is nothing more than changing the sign of the representing `SmallInteger` module $q^n - 1$. Discrete logarithms to base $\alpha$ can be read off directly from the representation and for computing the discrete logarithm to a arbitrary base one has to perform the extended euclidean algorithm in $\mathbb{Z}$.

If we want to compute the discrete logarithms $d$ of $\gamma = \alpha^m$ to base $\beta = \alpha^b$ we have to solve:

$$(\alpha^b)^d = \alpha^m$$

This is solvable if and only if $m$ is divisible by

$$g := \gcd(b, q^n - 1) = rb + s(q^n - 1)$$

. In this case we set

$$d = \frac{rm}{g} \bmod (q^n - 1)$$

Computing the multiplicative order of elements is done by

$$\mathrm{ord}(\alpha^e) = \frac{\mathrm{ord}(\alpha)}{\gcd(e, \mathrm{ord}(\alpha))} = \frac{q^n - 1}{\gcd(e, q^n - 1)}$$

Therefore an element $\beta$ is primitive in $E$ if and only if its representation is relatively prime to $q^n - 1$.

### 13.6.2 Addition and Zech logarithm

Addition is performed via the Zech (or Jacobi) logarithm $Z(k)$ which is defined by

$$\alpha^{Z(k)} = \alpha^k + 1$$

In the domain `E` the Zech logarithm array is stored in the global variable

```
zechlog : PrimitiveArray(SmallInteger)
```

of the datatype of length $(q^n + 1)/2$. Its $k$-th entry corresponds to $Z(k)$. $Z(k)$ is undefined, if $\alpha^k + 1 = 0$. This exception appears in characteristic 2 if $k = 0$ and in odd characteristic if $k = (q^n - 1)/2$. To indicate this we define `zechlog.k=-1` in these cases. Now the sum of $\alpha^i$ and $\alpha^j$ is computed in the following way:

Let $k$ be the smaller one of $\{(i - j) \bmod (q^n - 1),\ (j - i) \bmod (q^n - 1)\}$. Notice that $k \leq (q^n - 1)/2$. Then

$$\alpha^i + \alpha^j := \begin{cases} 0 & \text{if } Z(k) \text{ undefined} \\ \alpha^{i+Z(k)} & \text{if } k = (j - i) \bmod (q^n - 1) \\ \alpha^{j+Z(k)} & \text{if } k = (j - i) \bmod (q^n - 1) \end{cases}$$

In addition to some `SmallInteger` operatoins there is only one access to the Jacobi logarithm array. Therefore addition is very fast too.

Since $-1 = \alpha^{(q^n-1)/2}$ in odd characteristic, $-\alpha^n$ can be computed by

$$-\alpha^n := \begin{cases} \alpha^n & \text{if } \mathrm{char}(E) = 2 \\ \alpha^{n+(q^n-1)/2} & \text{otherwise} \end{cases}$$

The Jacobi logarithm array is initialized at the first time when it is needed. This procedure may last some time. The initialization is done by calling the function `createZechTable(f)` parameterized with the defining polynomial of the field in the package `FiniteFieldFunctions(F)`. The user gets access to this table by calling `getZechTable()$E`.

In K. Huber[Hube90] explains how to reduce the size of the table needed to compute all Jacobi logarithms. These observations are useful for high extension degree, but the times for addition are increased, therefore these ideas were not implemented.

To check if a given element belongs to $F$ is quite easy. It depends on whether the representation $e$ of $\alpha^e$ is divisible by $(q^n - 1)/(q - 1)$ or not. If $e = k(q^n - 1)/(q - 1)$ we have

$$(\alpha^e)^{q-1} = \alpha^{k(q^n-1)} = 1$$

and therefore $\alpha^e$ belongs to $F$. The degree of $\alpha^e$ over $F$ is given by the minimal integer $d > 0$ for which $eq^d \equiv e \bmod (q^n - 1)$. The trace and the norm function are computed directly using (1) on page 6.

### 13.6.3 Time expensive operations

But there remain some operations, which are quite time expensive. That are those operations, which change the representation of the field elements:

```
coerce:      F -> E
retract:     E -> F
represents:  Vector F -> E
coordinates: E -> Vector F
```

Let $\beta = N_{E/F}(\alpha)$ be the norm of $\alpha$ over $F$. Since $\beta$ is primitive in $F$, every nonzero element $\gamma$ of $F$ can be expressed by a power of $\beta$. We get

$$\gamma = \beta^e = \alpha^{e\frac{q^n-1}{q-1}}$$

for a suitable value $0 \leq e \leq q-1$. This $\beta$ is stored in the global variable `primEltGF` in `E`. The function `retract` applied to $(\alpha^e)$ first checks if the element $\alpha^e$ belongs to $F$. In this case $e$ is divisible by $(q^n-1)/(q-1)$ and `retract` can raise `primEltGF` in `F` to the power $e(q-1)/(q^n-1)$.

`coerce` applied to $\gamma \in F$, $\gamma \neq 0$, computes the discrete logarithm of $\gamma$ to the base `primEltGF` in `F` and multiples this value by $(q^n-1)/(q-1)$ to get the desired representation of $\gamma$ in `E`.

`coordinates` applied to $(\alpha^e)$ raises the residue class ($X$ mod $f$) to the power $e$. This is performed in a `SimpleAlgebraicExtension` of the `F` by `f`. The returned vector is the coordinate vector of $\alpha^e$ to the polynomial basis generated by $\alpha$.

`represents` considers the given vector as coordinate vector of an element $\beta$ in `FiniteFieldExtensionByPolynomial(F,f)` and computes its discrete logarithm to base $\alpha$ in that domain. This logarithm is the representation of $\beta$ in `E`.

## 13.7 Normal basis representation

Let $E = GF(q^n)$ be an extension of degree $n$ over the finite field $F = GF(q)$ and $f \in F[X]$ be the polynomial which defines the extension. Assume further that the roots $\{\alpha, \alpha^q, \ldots, \alpha^{q^{n-1}}\}$ of $f$ in $E$ are linearly independent over $F$. Then $\alpha$ is normal in $E$ over $F$ and every element of $E$ can be expressed in the form

$$\beta = \sum_{i=0}^{n-1} b_i \alpha^{q^i}$$

with $b_i \in F$. This kind of representation is used in the domains

```
FFNBP   FiniteFieldNormalBasisExtensionByPolynomial
FFNBX   FiniteFieldNormalBasisExtension
FFNB    FiniteFieldNormalBasis
```

Let `F` be a finite field domain representing $F$ and `E:=FFNBP(F,f)` the normal basis extension of $F$ by $f$.

We get the root $\alpha$ of $f$ in $E$ by calling `generator()$E` which in this representation is equal to `normalElement()$E`.

The internal representation of the elements of `E` is `Vector F`. The element $\beta = \sum_{i=0}^{n-1} b_i \alpha^{q^i}$ is represented by the coordinate vector $(b_0, b_1, \ldots, b_{n-1})$ of $\beta$ w.r.t. the normal basis generated by $\alpha$ and computed by `coordinates`($\beta$). The normal basis is returned by `basis()$E`. In

the sequel we identify coordinate vectors $(b_0, b_1, \ldots, b_{n-1})$ representing $\sum_{i=0}^{n-1} b_i \alpha^{q^i}$ with the corresponding polynomial $b = \sum_{i=0}^{n-1} b_i X^i \in F[X]/(X^n - 1)$, since

$$b \circ \alpha = \sum_{i=0}^{n-1} b_i \alpha^{q^i}$$

The lengthy code for the arithmetic is shared by the three different versions of normal basis representations and the package `FiniteFieldFunctions`. Hence, we decide to have a parameterized package

```
INBFF   InnerNormalBasisFieldFunctions(F)
```

where most of the arithmetic in `E` is performed.

### 13.7.1 Operations of additive nature

All field functions concerning the cyclic $F[X]$-module structure of the additive group of $E$ are very easy to implement and to compute. Since

$$\beta^q = \sum_{i=0}^{n-1} b_{(i-1) \bmod n} \alpha^{q^i}$$

the Frobenius automorphism becomes a simple cyclic shift of the coordinate vector. The linear associated logarithm of $\beta$ to base $\alpha$ can be directly read off from the representation of $\beta$

$$Log_\alpha(\beta) = \sum_{i=0}^{n-1} b_i X^i = b$$

To compute the linear associated logarithm $a$ of $\beta$ to another logarithm base $\gamma = c \circ \alpha$, one has to perform an extended euclidean algorithm in $F[X]$:

$$a \circ \gamma = \beta \Longleftrightarrow (ac) \circ \alpha = b \circ \alpha$$

This is solvable if and only if $b$ is divisible by $g := \gcd(c, X^n - 1) = rc + s(x^n - 1)$. In this case we get

$$a = \frac{rb}{g} \bmod (X^n - 1)$$

The operation $\circ$ becomes a modular polynomial multiplication

$$h \circ \beta = (hb) \circ \alpha$$

for $h \in F[X]$. The linear associated order of $\beta$ can be computed using

$$\mathrm{Ord}_q(b \circ \alpha) = \frac{\mathrm{Ord}_q(\alpha)}{\gcd(b, \mathrm{Ord}_q(\alpha))} = \frac{(X^n - 1)}{\gcd(b, X^n - 1)}$$

Therefore $\beta$ is normal in $E$ over $F$ if and only if $\gcd(b, X^n - 1) = 1$, which is quite easy to check. The degree of $\beta$ over $F$ is given by the minimal integer $d > 0$ which satisfies

$$bX^d \equiv b \bmod (X^n - 1)$$

The embedding of `F` into `E` is determined by the trace of $\alpha$: Let $a = T_{E/F}(\alpha)$, then $1 - T_{E/F}(a^{-1}\alpha) = \sum_{i=0}^{n-1} a^{-1}\alpha^{q^i}$ and we get for $d \in F$

$$d = \sum_{i=0}^{n-1} (a^{-1}d)\alpha^{q^i}$$

which gives as representation of $d$ in `E` the vector consisting of equal entries $a^{-1}d$. Since the value $T_{E/F}(\alpha)$ is needed quite often, it is stored in the global variable `traceAlpha` in `E`. The trace $T_{E/F}(\beta)$ of $\beta = \sum_{i=0}^{n-1} b_i\alpha^{q^i}$ is simply computed by

$$T_{E/F}(\beta) = \sum_{i,j=1}^{n-1} b_i\alpha^{q^{i+j}} = \sum_{j=0}^{n-1}(\sum_{i=0}^{n-1} b_i)\alpha^{q^i}$$

Traces onto intermediate fields of $F \leq E$ are computed in a similar fashion.

### 13.7.2 Multiplication and normal basis complexity

In the contrary to the *additive* functions, the operations concerning the multiplicative structure of the field are more difficult to compute. Actually the multiplication of field elements is somewhat complicated and hence slow.

To multiply field elements we use the representing matrix $M_\alpha = (m_{i,j}) \in F^{n\times n}$ of the left multiplication by $\alpha$ w.r.t. the distinguished normal basis, which is called *multiplication matrix* in Geiselmann/Gollmann [Geis89]:

$$\alpha\alpha^{q^i} = \sum_{j=0}^{n-1} m_{i,j}\alpha^{q^j}$$

Knowing this matrix the product of $\beta = \sum_{i=0}^{n-1} b_i\alpha^{q^i}$ and $\gamma = \sum_{j=0}^{n-1} c_j\alpha^{q^j}$ is given by

$$\beta\gamma = \sum_{i,j=0}^{n-1} b_i c_j(\alpha^{q^{i-j}+1})^{q^j} = \sum_{i,j,k=0}^{n-1} b_i c_j m_{i-j,k-j}\alpha^{q^k}$$

with indices module $n$. This shows immediately that multiplication in this representation needs $O(n^3)$ $F$-operations.

Recently there has been much interest in so called *low complexity* and *optimal* normal bases: By choosing the normal element $\alpha \in E$ carefully one tries to minimize the number of nonzero entries in $M_\alpha$, which is called the *complexity* of the normal basis. This obviously reduces the multiplication time. In the best case one can reduce the number of entries to $2n - 1$. For special pairs $(q, n)$ a direct construction of a normal base of low complexity $O(kn)$, $k \ll n$, is possible (see Wassermann [Wass89], Beth/Geiselmann/Meyer[Beth91], Mullin/Onyszchuk/Vanstone[Mull88] or Ash/Blake/Vanstone[Ashx89]). The problem of efficiently computing the minimal normal basis complexity or even a generator of such a base for given $(q, n)$ is unsolved.

The algorithm described by A. Wassermann in [Wass89] is implemented in the function

```
createLowComplexityNormalBasis(n)
```

in the package `FiniteFieldFunctions(F)`. If for the given $(q, n)$ a direct construction of a low complexity normal basis is possible, the algorithm computes the multiplication matrix of this base and the function returns this matrix in form of a variable of type `Vector List Record(value:F,index:SmallInteger)` (see below). If such a construction is not possible for $(q, n)$ the function

```
createNormalPoly(n)$FiniteFieldPolynomialPackage(F)
```

is called to produce a normal polynomial of degree $n$. To have the nice embedding $d \mapsto (d, d, \ldots, d)$ of `F` in `E`, the computed normal basis has in both cases the property that its generator has trace 1 over $F$. The constructors `FFNBX` and `FFNB` makes use of this function and use, if possible, automatically a low complexity normal basis.

If we would store the multiplication Matrix $M_\alpha$ in a variable of type `Matrix F`, everytime we multiply we would have to inspect all $n^2$ entries of $M_\alpha$. In this case a low complexity basis would hardly speed up the multiplication time.

This is why we store $M_\alpha$ in the domain $E$ in a global variable of the form

```
multTable : Vector List Record(value:F,index:SmallInteger)
```

The entry $m_{i,j}$ of $M_\alpha$ corresponds to the element of `multTable.i` with `index` $j-1$ and `value` $m_{i,j}$. Of course only the nonzero entries of $M_\alpha$ are stored in `multTable`. When multiplying now we are inspecting only the nonzero entries of $M_\alpha$ and get time advantages using bases of low complexity.

The first time when the multiplication matrix $M_\alpha$ is needed, it is initialized by an automatic call of the function `createMultiplicationTable(f)` in the package `FinitFieldFunctions(F)`.

The user has access to the multiplication matrix of the field by

```
getMultiplicationTable: () ->
     Vector List Record(value:F,index:SmallInteger)
  ++ getMultiplicationTable() returns the multiplication
  ++ table for the normal basis of the field
getMultiplicationMatrix: () -> Matrix F
  ++ getMultiplicationMatrix() returns the multiplication
  ++ table in the form of a matrix
```

The complexity of the normal basis can be found out by calling

```
sizeMultiplication:() -> NonNegativeInteger
```

### 13.7.3 Norm and multiplicative inverse

The functions `norm` and `inv` are the power functions with exponents $(q^n - 1)/(q - 1)$ and $(q^n - 2)$, respectively. We do not use the default repeated squaring algorithm, as we can do better: The algorithm due to Itoh and Tsujii[Itoh88] uses a clever partitioning of these special exponents and the following help function `expPot`. It computes

$$expPot(\beta, k, d) = \prod_{i=0}^{k-1} \beta^{q^{id}}$$

for $\beta \in E$ and integers $k, d > 0$ is computed by the (slightly simplified) algorithm

```
expPot(beta,k,d) ==
  e:Integer:=0
```

```
gamma:E:=1
for i in 0..length(k) repeat
  if bit?(k,i) then gamma:=gamma * beta**(q**e); e:=e+d
  beta:=beta * beta**(q**d); d:=2*d
return(gamma)
```

where `length(k)` denotes the number of bits of $k$, i.e. $\lceil log_2(k+1) \rceil$, and `bit?(k,i)` tests whether the $i$-th bit of $k$, binary represented, is set or not. The average number of $E$-multiplications of this algorithm is about $3/2 \lceil log_2(k) \rceil$

Let $d$ be a divisor of $n$ and $K$ and extension of degree $d$ over $F$. With the above algorithm we can compute the norm of $\beta \in E$ over $K$ by

$$N_{E/K}(\beta) = \text{expPot}(\beta, n/d, d)$$

using about $3/2 \lceil log_2(n/d) \rceil$ multiplications in $E$.

For computing the inverse $\beta^{-1} = \beta^{q^n-2}$ of $\beta \in E$ notice that

$$q^n - 2 = (q-2)(\frac{q^n-1}{q-1}) + q(\frac{q^{n-1}-1}{q-1})$$

therefore

$$\beta^{-1} = N_{E/F}(\beta)^{-1} \cdot (\beta^{\frac{q^{m-1}-1}{q-1}})^q$$

Now we get $\beta^{-1}$ by computing first

$$\gamma = \text{expPot}(\beta, n-1, 1)^q = (\beta^{\frac{q^{m-1}-1}{q-1}})^q$$

and then $\beta^{-1} = (\beta\gamma)^{-1} \cdot \gamma$. Notice that the inversion of $\beta\gamma$ is performed in $F$. Altogether we used $3/2 \lceil log_2(n) \rceil + 2$ multiplications in $E$.

### 13.7.4 Exponentiation

Next we show how we have implemented the function

$$(\beta, e) \to \beta^e$$

For a $1 \leq k < n$ we can write

$$\beta^e = \prod_i (\beta^{e_i})^{q^{ki}}$$

if $e = \sum_i e_i q^{ik}$ is the $q^k$-adic expansion of $e$.

An obvious implementation of this formula first of all has to initialize the array $[\beta, \beta^2, \ldots, \beta^{q^k-1}]$. This costs $(q-1)q^{k-1} - 1$ (expensive) field multiplications. Taking $q^{ki}$-powers is cheap while multiplying the results together costs another $\lceil (log_q(e)/k) \rceil - 1$ field multiplications, altogether this algorithm requires

$$M(q,k,e) := (q-1)q^{k-1} + \left\lceil \frac{log_q(e)}{k} \right\rceil - 2$$

multiplications.

Depending on $k$ there is a tradeoff between slow multiplication and fast powering. Therefore we adaptively choose a good $k$ depending on $e$ and $q$ to minimize the number of multiplications.

The computation of such $k \sim log_q log_q(e) - log_q log_q log_q(e)$ is performed using exclusively `SmallInteger` arithmetic to minimize the decision time. It is supported by the two global variables

```
logq:List SmallInteger
expTable:List List SmallInteger
```

which contain some precomputed auxiliary values.

Then the actual number of multiplications is compared with the number $3/2\,\lceil log_2(e) \rceil$ of multiplications needed by the standard repeated squaring algorithm and the better method is chosen.

The ideas of this divide and conquer algorithm are due to Stinson, see [Stin90], for the case $q = 2$.

## 13.8 Homomorphisms between finite fields

Changing an object from one finite field to another can be necessary in three cases. The first case is that we have two different defining polynomials for the same field in the same type of representations. In order to consider one object in the other data type we have to implement a field isomorphism. A generalization thereof is when one field is isomorphic to a subfield of another field in the same type of representation. The most complicated case is when in addition to the last situation we also change the representation.

These data type conversions - called *coercions* in Axiom, and hence are under the control of the interpreter - are realized by the package

```
FFHOM FiniteFieldHomomorphisms
```

It is parameterized by three parameters: a source field $K_1$ represented by `K1`, a target field $K_2$ represented by `K2` and a common groundfield $F$ of $K_1$ and $K_2$, represented by `F`. Note, that due to the symmetry of the provided functions, the order of the parameters can either be $(K1, F, K2)$ or $(K2, F, K1)$. The order comes from arranging the situation in a lattice.

However, this package cannot be used for the general situation as this settings suggests. We had to restrict ourselves to the case where both `K1` and `K2` are realized as simple extensions of `F`, i.e. of type

```
FiniteAlgebraicExtensionField(F)
```

To implement the general case also, it would be necessary to have a function

```
groundfield: () -> FiniteFieldCategory
```

Its result could be used for package calling functions of that subfield. Using such a function it would be possible to build a recursive coercion algorithm between different towers of finite field extensions. As the old compiler does not support functions whose values are domains, this idea will be possible when the new compiler will be available.

The source field and the destination field may appear in arbitrary order in the parametrization of `FFHOM`, since `FFHOM` supports coercions in both directions:

```
coerce: K1 -> K2
```

```
coerce: K2 -> K1
```

Restricted to $K_1 \cap K_2$ these two mappings are inverses of each other, i.e. for $\alpha \in K_1 \cap K_2$ and $a$ being a representation of $\alpha$ in `K1` or in `K2` holds:

```
coerce(coerce(a)$FFHOM(K1,F,K2))$FFHOM(K1,F,K2)=a
```

To be independent of the ordering of the arguments we have ordered the fields inside the package by comparing the defining polynomials of the fields lexicographically using the local function `compare`.

To explain the details of the implementation let $\beta \in K_1$ and `b` be the representation of $\beta$ in `K1`. We have to distinguish between some cases:

First check whether $\beta$ is in $F$. In this case

```
retract(b)@F$K1::K2
```

is used.

The next case is that `K1` and `K2` are constructed using the same defining polynomial $f$. If furthermore `K1` and `K2` are represented in the same say, the elements of `K1` and `K2` are represented completely identical. Therefore the coercion may be performed by

```
b pretend K2
```

Now assume that one of `K1` and `K2` is represented using cyclic groups and the other one represented by a polynomial basis.

If `K1` is cyclically represented, we coerce by

```
represents(coordinates(b)$K1)$K2
```

and vice versa, if `K2` is cyclically represented.

All remaining cases are treated in the same way which we explain now. Denote by `degree1` and `degree2` the extension degrees of $K_1$ and $K_2$ over $F$, respectively. The first time a coercion from `K1` into `K2`, or vice versa, is called, two conversion matrices stored in global variables

```
conMat1to2:Matrix F
  -- conversion Matrix for the conversion direction K1 -> K2
conMat2to1:Matrix F
  -- conversion Matrix for the conversion direction K2 -> K1
```

in the package are initialized. Once these matrices are initialized, the coercion is performed by

```
represents(conMat1to2 * coordinates(b)$K1)$K2
```

Notice, that we do not have to care about the coercion between cyclic representation and polynomial representation as above, since this step is implicitly performed by the function calls to `represents` and `coordinates` (see section 13.6.3).

The rest of this section describes the initialization of the conversion matrices.

### 13.8.1 Basis change between normal and polynomial basis representation

We first consider the case of equal defining polynomials. We can assume without loss of generality that $E = K_1 = K_2$ and the root $\alpha \in E$ of this polynomial both generates a polynomial and a normal basis. To convert $\beta = \sum_{i=0}^{n-1} b_i \alpha^i$ into $\beta = \sum_{j=0}^{n-1} c_j \alpha^{q^j}$ we have to

set up the basis change matrix $M = (m_{i,j})$, defined by

$$\alpha^{q^j} = \sum_{i=0}^{n-1} m_{i,j}\alpha^i, \quad 0 \le j < n$$

Then we have

$$\beta = \sum_{j=0}^{n-1} c_j \alpha^{q^j} = \sum_{i,j=0}^{n-1} c_j m_{i,j} \alpha^i$$

consequently

$$b_i = \sum_{j=0}^{n-1} c_j m_{i,j}$$

Therefore $M$ can be used to change between polynomial representation and normal representation by

$$M(c_0, c_1, \ldots, c_{n-1})^t = (b_0, b_1, \ldots, b_{n-1})^t$$

and

$$M^{-1}(b_0, b_1, \ldots, b_{n-1})^t = (c_0, c_1, \ldots, c_{n-1})^t$$

where the $t$ denotes the transposition of the vector. The matrix $M$ is computed efficiently by using the function `reducedQPowers` in the package `FiniteFieldPolynomialPackage(F)`.

### 13.8.2 Conversion between different extensions

Now suppose that `K1` and `K2` are built using different defining polynomials `defpol1` and `defpol2`, respectively. As noticed earlier we have to order these polynomials to get the same homomorphisms when calling `FFHOM` with parameters `K1` and `K2` swapped.

Without loss of generality let `defpol1` be the lexicographical smaller polynomial of `defpol1` and `defpol2`. In the other case '1' and '2' are swapped in what follows. Let $n_1$ and $n_2$ be the extension degrees of $K_1$ and $K_2$ over $F$, respectively. Note that $n_2 \ge n_1$.

We first compute a root $\alpha_1 = \sum_{i=0}^{n_2-1} a_i \alpha_2^i$ of `defpol1` in a polynomial basis representation `FFP(F,defpol2)` of $K_2$ using

```
rootOfIrreduciblePoly(defpol1)$FFPOLY2(FFP(F,defpol2),F)
```

By powering this root we compute the Matrix $R = (r_{i,j}) \in F^{n_2 \times n_1}$ defined by

$$\alpha_1^j = \sum_{i=0}^{n_2-1} r_{i,j}\alpha_2^i \quad 0 \le j < n_1$$

If $\beta = \sum_{j=0}^{n_1-1} b_j \alpha_1^j \in K_1$ we get

$$\beta = \sum_{j=0}^{n_1-1} \sum_{i=0}^{n_2-1} b_j r_{i,j} \alpha_2^i = \sum_{i=0}^{n_2-1} c_i \alpha_2^i$$

where $c_i = \sum_{j=0}^{n_1-1} b_j r_{i,j}$. In the situation where both representations are polynomial it is enough to use $R$ for converting elements from $K_1$ into $K_2$ by

$$(c_0, c_1, \ldots, c_{n_2-1})^t = R(b_0, b_1, \ldots, b_{n_1-1})^t$$

To construct a concrete matrix representation $S$ of a left inverse linear map of the $F$-Monomorphism represented by $R$ we proceed as follows. We construct a square matrix $P$ by taking the first $n_1$ linearly independent rows of $R$, invert $P$ and put zero columns at the proper places of $P$ to get $S \in F^{n_1 \times n_2}$.

If one or both representations are normal, we use the basis change matrices from section 13.8.1 to reduce this case to the polynomial case. More precisely, if $K_2$ is normal basis represented we construct a basis change matrix $M_2 \in F^{n_2 \times n_2}$ and compute

$$R := M_2^{-1}R, \quad S := SM_2$$

If $K_1$ is normal basis represented we construct a basis change matrix $M_1 \in F^{n_1 \times n_1}$ and compute

$$R := RM_1, \quad S := M_1^{-1}S$$

.

After this computation $R$ and $S$ are the desired conversion matrices

```
conMat1to2:=R
conMat2to1:=S
```

Now we can coerce from `K1` into `K2` by

```
represents(conMat1to2 * coordinates(b)$K1)$K2
```

and in the other direction by

```
represents(conMat2to1 * coordinates(b)$K2)$K1
```

## 13.9 Polynomials over finite fields

Let $F = GF(q)$ and $E = GF(q^n)$ be represented by the domain `F` and `E`, respectively. There are two packages with functions concerning polynomials over finite fields.

| | |
|---|---|
| `FFPOLY(F)` | `FiniteFieldPolynomialPackage(F)` |
| `FFPOLY2(E,F)` | `FiniteFieldPolynomialPackage2(E,F)` |

### 13.9.1 Root finding

Although Axiom has a good factorizer for polynomials over fields, which can be used to find roots in extensions, we have implemented a function

```
rootOfIrreduciblePoly(f)
```

in the package `FFPOLY2(E,F)`. This function computes only one root in $E$ of a monic, irreducible polynomial $f \in F[X]$. This is useful as in section 13.8 we have described how the knowledge of one root is enough to find homomorphisms between finite fields by the package `FFHOM`. Furthermore, this implementation is in general faster than the whole factorization. It uses Berlekamp's algorithm as described in chap.3.4 of [Lidl83].

### 13.9.2 Polynomials with certain properties

In `FFPOLY(F)` there are functions concerning the properties *irreducibility*, *primitivity*, *normality* of monic polynomials. Furthermore, Lenstra and Schoof proved in [Lens87] the exis-

tence of polynomials of arbitrary degree, which are both primitive and normal for arbitrary finite of finite fields $F$.

Hence, we have four kinds of properties. We have implemented functions to check, whether a given polynomial is of a certain kind, create a polynomial of a given degree of a certain kind, search the next polynomial of a certain kind w.r.t. an ordering that will be described below, and compute the number of polynomials of a given degree of a certain kind.

The functions for normal polynomials are for example

```
normal?(f)
createNormalPoly(n)
nextNormalPoly(f)
numberOfNormalPoly(n)
```

Unfortunately, there is no formula known to compute the number of primitive and normal polynomials of a given degree. Up to this exception similar operations are available for all four kinds of properties. To get these functions one has to substitute `normal` by `irreducible`, `primitive`, or `normalPrimitive` (or equivently `primitiveNormal`), respectively.

### 13.9.3 Testing whether a polynomial is of a given kind

The function `irreducible?` is in the package `DistinctDegreeFactorize`, where it had been implemented for polynomial factorization purposes. It uses the fact that a polynomial $f \in F[X]$ of degree $n$ is irreducible, if and only if $f$ is relatively prime to $X^{q^i} - X$ for all $i = 1, 2, \ldots, \lfloor n/2 \rfloor$. This and two other methods for testing irreducibility are described in a nice way in A.K. Lenstra's paper[Lens82].

To check whether a polynomial $f \in F[X]$ of degree $n$ is primitive we use the algorithm described on page 87 in [Lidl83] together with theorem 3.16. It is the same algorithm as used for testing elements of finite field domains on primitivity:

$f$ is primitive if and only if $X^{q^n-1} \bmod f = 1$ and if for all prime factors $p$ of $q^n - 1$ holds:

$$X^{(q^n-1)/p} \bmod f \neq 1$$

To check whether a polynomial $f \in F[X]$ of degree $n \geq 1$ is normal, we first check its irreducibility and compute then the residues

$$X^{q^i} \bmod \mathrm{f}, \quad \text{for } 0 \leq i < n$$

using the function `reducedQPowers(f)`. Now $f$ is normal if and only if this $n$ residues are linearly independent over $F$.

To check whether a polynomial is primitive and normal we have to combine both tests.

### 13.9.4 Searching the next polynomial of a given kind

In this section we describe various total orderings on the set of monic polynomials $f = X^n + \sum_{i=0}^{n-1} f_i X^i \in F[X]$ of degree $n$ with constant nonzero term $f_0$. These orderings are chosen carefully to reflect special requirements on embedding and constructing properties and sparsity for the various kinds.

Let $\#f$ be the number of nonzero coefficients of $f$ and terms($f$) denote the vector $(i : f_i \neq 0)$ of exponents appearing in $f$ in ascending order. The core of all our orderings is the following

total order. If $g = X^n + \sum_{i=0}^{n-1} g_i X^i$ is another element in $F[X]$, then $f < g$, if $f$ is more sparse than $g$, i.e. $\#f < \#g$. This is very useful, as dealing with sparse polynomials in polynomially represented finite field arithmetic is less time consuming, see section 13.11. The ordering then is refined by the lexicographical ordering of the exponent vectors terms($f$) and terms($g$), and, if they are equal, too, by the lexicographical ordering of the coefficient vectors, induced by the comparison of elements in $F$ via `lookup`, i.e. for elements `fi`, `gi` of `F`, `fi` $<$ `gi` if and only if `lookup(fi)` $<$ `lookup(gi)`.

For the irreducibility we directly use this ordering. `nextIrreduciblePoly(f)` returns the next irreducible polynomial of degree $n$ which is greater than $f$.

The ordering for `nextPrimitivePoly(f)` first compares the constant terms. If $f_0 = g_0$, then we use the ordering defined above. This is for efficiency reasons. Changing $f_0$ requires the computation of the next (w.r.t `lookup`) primitive element of $F$. This procedure takes more time than changing other coefficients and is therefore done last.

Similarly, for normal polynomials we first pick the trace coefficients $f_{n-1}$ and $g_{n-1}$ and compare these, if they are equal we compare as described above. So we have tied together all those normal polynomials, which define the same embedding of the ground field.

For `nextPrimitiveNormalPoly(f)` we order first according to the constant terms, then according to the trace terms and finally as above.

All of the functions yield `"failed"` when called with a polynomial of degree $n$, which is greater than the greatest polynomial with the required property.

### 13.9.5 Creating polynomials

To create an irreducible polynomial Axiom proceeds as follows. Depending on the desired degree $n$, a start polynomial $f$ is initialized with one value of $\{X^n, X^n + 1, X^n + X\}$. Then `nextIrreduciblePoly(f)` is called to produce an irreducible polynomial.

If $f(X) = X^n + \sum_{i=0}^{n-1} f_i X^i \in F[X]$ is primitive and $\alpha$ is a root of $f$ in $E$ then $N_{E/F}(\alpha) = (-1)^n f_0$ is primitive in $F$. Since the norm function is surjective, there exist primitive polynomials $f$ for all $(-1)^n f_0$ which are primitive in $F$. Therefore, to get a primitive polynomial $f_0$ is set to $(-1)^n$ times `primitiveElement()$F`. Then the polynomials with $(f_1, \ldots, f_{n-1})$ ranging over $\{0,1\}^{n-1}$ are checked on primitivity in increasing order. The used ordering of $\{0,1\}^{n-1}$ is ordering of the vectors by (hamming) weight, where the vectors of equal weight are ordered lexicographically. If no primitive polynomial is found by this procedure `nextPrimitivePoly`$(X^n + f_0)$ is called to produce one.

If $f(X) = X^n + \sum_{i=0}^{n-1} f_i X^i \in F[X]$ is normal over $F$ and $\alpha$ is a root of $f$ in $E$ then $T_{E/F}(\alpha) = (-f_{n-1}) \neq 0$. Since the trace function is surjective, there exist normal polynomials $f$ for all choices of $0 \neq f_{n-1} \in F$. We set $f_{n-1} = -1$, thus $T_{E/F}(\alpha) = 1$ and we get a nice embedding of $F$ into $E$ (see section 13.7.1). A normal polynomial is generated by calling `nextNormalPoly`$(X^n - X^{n-1})$.

Normal polynomials which are primitive, too, are constructed analogically. We set $f_{n-1} = -1$ and $f_0 = (-1)^n c$, where $c$ =`primitiveElement()$F`. Then `nextNormalPrimitivePoly`$(X^n - X^{n-1} + f_0)$ is called.

### 13.9.6 Number of polynomials of a given kind and degree

Formulae for the number of polynomials of the above kinds can be found in many textbooks about finite fields, e.g. [Lidl83]

The number of monic irreducible polynomials in $F[X]$ of degree $n$ is given by

$$\frac{1}{n}\sum_{d|n}\mu(n/d)q^d$$

where $\mu$ denotes the number-theoretical Möbius function on $(\mathbb{N},|)$, see Theorem 3.25 in [Lidl83].

The number of primitive polynomials in $F[X]$ of degree $n$ is 1 if $q=2$ and $n=1$. Otherwise it is given by

$$\frac{1}{n}\varphi(q^n-1)$$

where $\varphi$ denotes the Euler $\varphi$-function, see Theorem 3.5 in [Lidl83].

The number of normal polynomials in $F[X]$ of degree $n$ is given by

$$\frac{1}{n}q^n\prod_{n_i}(1-q^{-n_i})$$

where the product ranges over the degrees $n_i$ of the distinct monic irreducible polynomials appearing in the canonical factorization of $(X^n-1)$ in $F[X]$, see Theorem 3.73 in [Lidl83].

### 13.9.7 Some other functions concerning polynomials

There are two functions for generating random polynomials.

`random(n)` yields a monic random polynomial of degree $n$. For the random selection of the coefficients the function `random()$F` from `F` is used.

The function `random(n,m)` yields a random polynomial $f$ of degree $n \le deg(f) \le m$. After determining a random degree $d$ by using the function `random()$Integer` the polynomial is computed by calling `random(d)`.

The least affine multiple of a polynomial $f$ is defined to be the polynomial $g \in F[X]$ of least degree of the form

$$g=\sum_i g_i X^{q^i}+\tilde{g},\quad g_i,\tilde{g}\in F_q$$

which is divisble by $f$ – remember that $q$ is the order of $F$!. The function call `leastAffineMultiple(f)` computes the least affine multiple of $f$ using the algorithm describe on page 112 of [Lidl83].

Let $f \in F[X]$ be monic irreducible of degree $n$ and $\alpha$ be a root of $f$ in $E$. The matrix $Q \in F^{n\times n}$ defined by

$$\alpha^{q^j}=\sum_{i=0}^{n-1}q_{i,j}\alpha^i,\quad 0\le j<n$$

plays an important role in many computations concerning finite fields (see e.g. section 13.8.1). Therefore we implemented the function `reducedQPowers(f)`, which yields the array

$$[X \bmod f, X^q \bmod f, \ldots, X^{q^{n-1}} \bmod f]$$

of $n$ polynomials. This array is computed efficiently using the fact that powering by $q$ is a $F$-linear operation.

## 13.10 Future directions

As already mentioned in the text the new compiler will open much more flexibility with datatypes. This could be used for an implementation which realizes each finite field datatype as an extension of each of its subfields.

The use of domain-valued functions as e.g. `groundField` would allow more general extension mechanisms which in particular could be used for recursive `coerce`-functions between different towers of extensions and representations.

Next we discuss the problem of better embedding between different extensions for special representations. In the case of cyclic group representation J.H. Conway has suggested to use *norm-compatible* defining polynomials, which were called *Conway*-polynomials (satisfying an additional condition) by R.A. Parker (see [Nick88]). These polynomials are primitive polynomials, whose roots are related to each other by the norm functions. Such classes of polynomials are difficult to compute. The pay-off of this computations are easy embeddings by integer multiplications.

The second author has developed a theory of *trace-compatible* defining polynomials for the embedding of normal basis representation of the extensions. The resulting embedding are polynomials multiplications in $F[X]$. He has implemementations of both these methods in Axiom and furthermore he has used these for an implementation of different representations of the algebraic closure of a finite field. Contrary to all the other datatypes, which became part of the Axiom system, these domain constructors are right now not generally available. We hope that finally there will be a way of distribution of this code.

A detailed description of both Conway's and the trace-compatibility method as well as the further implementations can be found in the *Doktorarbeit* [Sche93] of the second author, see also [Lidl83].

## 13.11 Comparison of computation times between different representations

To demonstrate the effects of different representations on time efficiency, we added tables with computation times for representative functions of some finite fields. The tables show the computation times of these functions in milliseconds. The times were yield in an Axiom session on an IBM RISC System/6000 model 550 workstation using the Axiom system command `)set message time on`. Each command was run 50 to 100 times for random values and the measured time was divided by the number of runs.

### 13.11.1 The extension fields $GF(5^4)$ over $GF(5)$ and $GF(2^{10})$ over $GF(2)$

We first consider two small fields:

Table 1: Computation times for $GF(2^{10})$ and $GF(5^4)$

| | $GF(2^{10})$ | | | $GF(5^4)$ | | |
|---|---|---|---|---|---|---|
| operation | poly | normal | cyclic | poly | normal | cyclic |
| addition | 1 | 1 | 1 | 1 | 1 | 1 |
| multiplication | 4 | 18 | 1 | 3 | 4 | 1 |
| inversion | 13 | 105 | 1 | 10 | 16 | 1 |
| primitive? | 71 | 158 | 1 | 28 | 41 | 1 |
| normal? | 88 | 4 | 15 | 25 | 4 | 5 |
| minimalPolynomial | 130 | 290 | 28 | 33 | 44 | 10 |
| degree | 92 | 3 | 1 | 17 | 2 | 1 |
| discreteLog | 150 | 440 | 1 | 110 | 200 | 1 |
| norm | 10 | 106 | 1 | 6 | 17 | 1 |
| associateLog | 67 | 1 | 120 | 25 | 1 | 50 |
| associateOrder | 150 | 5 | 310 | 49 | 4 | 100 |
| associateEsp | 22 | 4 | 2 | 29 | 4 | 15 |
| trace | 16 | 5 | 3 | 5 | 3 | 3 |
| exponentiation | 37 | 70 | 1 | 20 | 21 | 1 |

$GF(2^{10})$ was built up as an extension of $GF(2)$ with the polynomial $X^{10}+X^9+x^4+X+1$ which is primitive and normal over $GF(2)$ at the same time.

An extension $GF(5^4)$ of $GF(5)$ via the primitive and normal polynomial $X^4+4X^3+X+2$ over $GF(5)$.

Table 1 shows the computation times for the three different representations of both fields.

### 13.11.2 Different extensions of $GF(5^{21})$ over $GF(5)$

For fields with many elements we omit the cyclic group representation. In extensions of high degree the computation time depends on whether one takes a 'good' or 'bad' polynomial for the representation. 'Good' in the normal basis representation means a low complexity for the according normal basis while 'good' in the polynomial basis representation means a small number of nonzero coefficients of the polynomial.

Table 2: Computation times for $GF(5^{21})$

| | 'good' | | 'bad' | |
|---|---|---|---|---|
| operation | polynomial | normal | polynomial | normal |
| addition | 1 | 1 | 1 | 1 |
| multiplication | 35 | 86 | 34 | 188 |
| inversion | 95 | 710 | 95 | 3040 |
| trace over GF(5) | 206 | 14 | 208 | 17 |
| norm over GF(5) | 78 | 606 | 78 | 2660 |
| minimalPolynomial | 1370 | 2590 | 1360 | 9000 |
| associateOrder | 2780 | 35 | 2700 | 34 |
| associateLog | 710 | 1 | 1500 | 1 |
| associateEsp | 1140 | 28 | 1690 | 20 |
| exponentiation with exponents in range | | | | |
| $<100$ | 210 | 350 | 250 | 1570 |
| $<1000$ | 420 | 510 | 420 | 2200 |
| $<10000$ | 590 | 600 | 600 | 2700 |
| $\sim 5^{21}$ | 2400 | 1600 | 2500 | 7400 |

We examine the field extension $GF(5^{21})$ over $GF(5)$. In the normal basis representation we chose the 'good' polynomial

$$X^{21}+X^{20}+X^{18}+X^{17}+3X^{16}+4X^{15}+2X^{11}+2X^{10}+3X^8+3X^7+4X^6+2X^5+X+1 \quad (2)$$

which yields a normal basis of low complexity of 61 and the 'bad' polynomial

$$X^{21}+4X^{20}+1$$

which yields a normal basis complexity of 323.

For the polynomial basis representation we chose as the 'bad' polynomial (2) and the 'good' polynomial $X^{21}+4X+1$. The table2 shows the results.

Although the multiplication time is much higher in the normal basis representation, the adaptive exponentiation algorithm yields for high exponents lower exponentiation times.

## 13.12 Dependencies between the constructors

The picture on the next page visualises the dependencies between the different constructors of the finite field world of Axiom.

Here is the list of used abbreviations:

**Categories**

| | |
|---|---|
| CHARNZ | CharacteristicNonZero |
| FPC | FieldOfPrimeCharacteristic |
| XF | ExtensionField |
| FFC | FiniteFieldCategory |
| FAXF | FiniteAlgebraicExtensionField |

**Domains**

| | |
|---|---|
| SAE | SimpleAlgebraicExtension |
| IPF | InnerPrimeField |
| PF | PrimeField |
| FFP | FiniteFieldExtensionByPolynomial |
| FFCGP | FiniteFieldCyclicGroupExtensionByPolynomial |
| FFNBP | FiniteFieldNormalBasisExtensionByPolynomial |
| FFX | FiniteFieldExtension |
| FFCGX | FiniteFieldCyclicGroupExtension |
| FFNBC | FiniteFieldNormalBasisExtension |
| IFF | InnerFiniteField |
| FF | FiniteField |
| FFCG | FiniteFieldCyclicGroup |
| FFNB | FiniteFieldNormalBasis |

**Packages**

| | |
|---|---|
| DLP | DiscreteLogarithmPackage |
| FFF | FiniteFieldFunctions |
| INBFF | InnerNormalBasisFieldFunctions |
| FFPOLY | FiniteFieldPolynomialPackage |
| FFPOLY2 | FiniteFieldPolynomialPackage2 |
| FFHOM | FiniteFieldHomomorphisms |

# Chapter 14

# Real Quantifier Elimination Tutorial by Hoon Hong

This material is quoted from Hoon Hong's presentation at Kyushu University in Japan in 2005. The state of the art has moved a bit but this is an excellent presentation up to that date.[Hong05]

## 14.1 Overview

**What is the real quantifier elimination problem?** As a simple example, consider the following "toy" problem: Find a condition on $b$ and $c$ such that $x^2 + bx + c > 0$ for all $x$. Recalling middle school math, we know that an answer is $b^2 - 4c < 0$. A bit more formally, one can write the above process as

In: $(\forall x) \quad x^2 + bx + c > 0$

Out: $b^2 - 4c < 0$

Note that the input formula and the output formula are equivalent but the output formula does not have the universal quantifier $(\forall x)$. We have just carried out the "real quantifier elimination". Generally, it is a problem:

In: a formula (made of integral polynomials, $=$, $>$, $\wedge$, $\vee$, $\neg$, $\forall$, $\exists$), the so-called "a well-formed formula in the first order theory of real closed field".

Out: an equivalent formula *without* quantifiers.

Note that the decision problem (proving) is a special case where the input formula does not have free variables (since it is trivial to decide a formula without variables). The research goal, then, is to devise efficient algorithms/software that carry out the real quantifier elimination.

**Why work on this problem?** There are two main sources of motivation for tackling the quantifier elimination problem: the original motivation from the foundational questions of mathematics, and the more recent motivation from applications in various areas.

The original motivation was based on the observation that the "existence" of a quantifier elimination procedure often implies various other important properties (such as completeness) about the theory under investigation and other theories that can be reduced to it.

On the other hand, the more recent motivation is based on the observation that real quantifier eliminating provides a simple but expressive abstraction for various important problems arising from constructive mathematics, and in particular various non-trivial problems in science and engineering, such as geometric modeling, geometric theorem proving and discovery, termination proof of term rewrite systems, stability analysis of control systems or numerical schemes for partial differential equations, approximation theory, optimization, constraint logic programming, fracture mechanics, robot motion planning, etc. Thus, progress in real quantifier elimination can have significant impact on algorithmic (computer-assisted) mathematics, science and engineering.

**What has been done so far?** Alfred Tarski[Tars48], around 1930, gave the first algorithm for the problem. This was an epoch making event in the foundational studies in mathematics and logic because it meant that not only the elementary real algebra but also various mathematical theories built on real numbers are also decidable, for instance, the elementary algebra of complex numbers, that of quarternions, that of n-dimensional vectors, and elementary geometry (Euclidean, non-Euclidean, projective).

Since then, various improved, new, and specialized algorithms have been devised with better computing times for large inputs (asymptotically) or for small inputs or for special inputs. Now computer implementations exist and are applied to various problems from mathematics, sicence and engineering.

**References** There are several collections dedicated to quantifier elimination (as of 2005):

- *Algorithms in Real Algebraic Geometry.*[Arno88a]
- *Computational Quantifier Elimination* Hoon Hong, Oxford University Press (1993)
- *Quantifier Elimination and Cylindrical Algebraic Decomposition* [Cavi98]
- *Quantifier Elimination and Applications* Journal of Symbolic Computation, Hoon Hong and Richard Liska, Academic Press (1996)

There are also a few books that give and exposition on the subject

- *Algorithmic Algebra*[Mish93]
- *Goemetrie algebrique reelle* Jacek, Bochnak, Michel Coste, Marie-Francoius Roy (1986)

## 14.2 General Methods

We begin by describing several general methods that work for arbitrary formulas. We will start with the historically first method (by Tarski[Tars48]), then proceed to the next important method of cylindrical algebraic decomposition (by Collins[Coll75]), then conclude with recent methods with better asymptotic computing bounds.

### 14.2.1 The First Method

Alfred Tarski, the Polish-born U.S. mathematician, logician and philosopher, discovered the first quantifier elimination algorithm about 1930, and submitted a monograph for publication in 1939 which was scheduled to appear in 1939 under the title *The completeness of elementary algebra and geometry* in the collection *Autualités scientifiques et industrielles*. However due

to the World War, the publication did not materialize. In 1948, the monograph was rewritten by his friend J. C. C. McKinsey and was published under the title *A Decision Method for Elementary Algebra and Geometry*. In 1967, Tarksi's original monograph was also published by the Institute Blaise Pascal in Paris.

The method of Tarski is not efficient. The worst case asymptotic computing time is *non-elementary* in that it cannot be bounded by an finite tower of exponential functions in the number of variables. But it contains several important ideas/techniques that had significant influence on the later methods.

### Method

Tarskis method carries out a series of reduction steps until it comes down to some simple cases which are tackled using Sturms theorem[Stur1829] and its generalization.

**Reduction 1**: First, the general quantifier elimination problem can be easily reduced to that for the formulas with *one existential* quantifier. This is because we can rewrite universal quantifiers in term of existential quantifiers ($\forall x\phi$ is equivalent to $\neg\exists x\neg\phi$) and then repeatedly eliminate the innermost existential quantifier one at a time.

**Reduction 2**: Now, we need to eliminate the quantifier from a formula of the form $\exists x\phi$. By pushing the negations inward onto the relational operators, rewriting $\geq$ in terms of disjunction of $=$ and $>$, putting the resulting formula into disjunctive normal form, and by using that the fact that disjunction commutes with an existential quantifier, we can reduce the problem to the following three types:

(1) $\exists x[P_1 = 0 \wedge \cdots \wedge P_r = 0]$
(2) $\exists x[P_1 = 0 \wedge \cdots \wedge P_r = 0 \wedge Q_1 > 0 \wedge \cdots \wedge Q_s]$
(3) $\exists x[Q_1 > 0 \wedge \cdots \wedge Q_x > 0]$

**Reduction 3**: Using the fact that $P_1 = 0 \wedge \cdots \wedge P_r = 0$ is equivalent to $P_1^2 + \ldots + P_r^2$, we can reduce the above (1) and (2) to

(1′) $\exists x[P = 0]$
(2′) $\exists x[P = 0 \wedge Q_1 > 0 \wedge \cdots \wedge Q_s]$

**Reduction 4**: Next we will reduce the case (3) to the case (2′). Suppose that all the polynomials $Q_i$'s are positive on some value $p$ of $x$. Then there are only three possibilities:

- All the polynomials continued to be positive for $x > p$.
- All the polynomials continued to be positive for $x < p$
- There are two values $p_1$ and $p_2$, $p_1 < p < p_2$ such that one of the polynomials vanish on $p_1$ and one (possibly the same) of the polynomials vanish on $p_2$. More succinctly put, the product of all the polynomials vanishes on $p_1$ and $p_2$. By Rolle's theorem[Roll1691], then, there must exist in $(p_1, p_2)$ a real root of the derivative of the product of the

polynomials.

Let lc denote leading coefficient. We can write the above case analysis formally:

- $\mathrm{lc}(Q_1) > 0 \wedge \cdots \wedge \mathrm{lc}(Q_s) > 0$
- $\mathrm{lc}(Q_1)(-1)^{\deg(Q_1)} > 0 \wedge \cdots \wedge \mathrm{lc}(Q_s)(-1)^{\deg(Q_s)} > 0$
- $\exists x[(Q_1 \cdots Q_s)' = 0 \wedge Q_1 > 0 \cdots Q_s > 0]$

Note that the first two are already quantifier-free and the last one belongs to the case $(2')$. Thus, the case (3) has been reduced to the case $(2')$.

Actually the reasoning above is not entirely correct. It can happen that a (formal) leading coefficient might depend on free variables, and for some values of the free variables, the leading coefficient might vanish, making it no longer the leading coefficient. In order to correct it, we will have do some more easy but tedious case analysis depending on the vanishing of the coefficients.

**Reduction 5**: Let $\exists_k x\phi$ means that there exactly $k$ distinct real values of $x$ that satisfy $\phi$. Using this notation, we can reduce the case $(1')$ and $(2')$ to the following:

$(1'')$ $\quad \exists_0 x[P = 0]$

$(2'')$ $\quad \exists_0 x[P = 0 \wedge Q_1 > 0 \wedge \cdots \wedge Q_s]$

This is because $\exists x\phi$ is equivalent to $\neg\exists_0 x\phi$. The reason for such rewriting is to prepare for a certain induction done in the next reduction.

**Reduction 6**: We will reduce the case $(2'')$ to the following:

$(2''')$ $\quad \exists_k x[P = 0 \wedge Q > 0]$

We will do this by repeatedly decreasing the number of inequalities one at a time. For this,

we only need to observe the following. Let

$$\begin{aligned}
n_{++} &= \#\{x|P=0\wedge Q_1>0\wedge\cdots\wedge Q_{s-2}>0 \\
&\qquad \wedge Q_{s-1}>0\wedge Q_s>0\} \\
n_{+-} &= \#\{x|P=0\wedge Q_1>0\wedge\cdots\wedge Q_{s-2}>0 \\
&\qquad \wedge Q_{s-1}>0\wedge Q_s<0\} \\
n_{-+} &= \#\{x|P=0\wedge Q_1>0\wedge\cdots\wedge Q_{s-2}>0 \\
&\qquad \wedge Q_{s-1}<0\wedge Q_s>0\} \\
r_1 &= \#\{x|P=0\wedge Q_1>0\wedge\cdots\wedge Q_{s-2}>0 \\
&\qquad \wedge Q_{s-1}Q_s^2>0\} \\
r_2 &= \#\{x|P=0\wedge Q_1>0\wedge\cdots\wedge Q_{s-2}>0 \\
&\qquad \wedge Q_{s-1}^2Q_s>0\} \\
r_3 &= \#\{x|P=0\wedge Q_1>0\wedge\cdots\wedge Q_{s-2}>0 \\
&\qquad \wedge Q_{s-1}Q_s>0\}
\end{aligned}$$

Then we have the following

$$\begin{aligned}
r_1 &= n_{++}+n_{+-} \\
r_2 &= n_{++}+n_{-+} \\
r_3 &= n_{+-}+n_{-+}
\end{aligned}$$

By solving for $n_{++}$, we immediately obtain

$$n_{++}=\frac{r_1+r_2-r_3}{2}$$

From this observation, we see that the formula

$$\exists_k x[P=0\wedge Q_1>0\wedge\cdots\wedge Q_s]$$

is equivalent to the formula with one less inequalities

$$\bigvee_{k=\frac{r_1+r_2-r_3}{2}} \left\{ \begin{array}{c} \exists_{r_1}x[P=0\wedge Q_1>0\wedge\cdots\wedge Q_{s-2}>0 \\ \wedge\ Q_{s-1}^2Q_s>0]\ \wedge \\ \exists_{r_2}x[P=0\wedge Q_1>0\wedge\cdots\wedge Q_{s-2}>0 \\ \wedge\ Q_{s-1}Q_s^2>0]\ \wedge \\ \exists_{r_3}x[P=0\wedge Q_1>0\wedge\cdots\wedge Q_{s-2}>0 \\ \wedge\ Q_{s-1}Q_s>0] \end{array} \right\}$$

By inductively applying the same trick on the above three subformulas we can arrive at the case $(2''')$.

**Summary of all reductions:** Through all the above reduction steps, the general quantifier elimination problem is reduced to the following two special cases:

$(1''')$ $\exists_k x[P=0]$
$(2''')$ $\exists_k x[P=0\wedge Q>0]$

The case $(1''')$ is a bit more general than the case $(1'')$ that we actually need. But we will keep it that way since it is symmetric to the case $(2''')$. From now on, we will tackle these two special cases.

**Case** $(1''')$ This case can be readily handled by *Sturm's theorem*[Stur1829]. Let $P$ be a univariate polynomial. Let $P_1 = P, P_2 = P', P_3, \ldots, P_n$ be a sequence where $P_{k+1}$ is the negative of the remainder obtained by dividing $P_{k-1}$ by $P_k$, and $P_n$ is the last non-zero polynomial in the sequence. Let $\alpha$ be the number of changes of sign in the sequence for $x \to -\infty$ and let $\beta$ be that for $x \to \infty$. Then Sturm proved that the number of distinct real roots of $P$ is exactly $\alpha - \beta$.

When there are no free variables in the formula $(1''')$, we only need to apply the Sturms method to see if the number of distinct real roots is indeed $k$. When there are free variables, we need to parametrize the Sturms method. We will have to do some easy but tedius case analysis depending the signs of the leading coefficients.

**Case** $(2''')$ This case can be handled again readily by Sylvester's generalization of Sturm's theorem. Let $P$ and $Q$ be two univariate polynomials and let $P_1 = P, P_2 = P'Q, P_3, \ldots, P_n$ be a sequence obtained the same way as in Sturm's method. Note that the only difference is that $P'Q$ is used in place of $P'$. Then Sylvester proved that

$$\alpha - \beta = \#\{x | P = 0 \wedge Q > 0\} - \#\{x | P = 0 \wedge Q < 0\}$$

Let $N(P, Q)$ denote this number. Further let

$$\begin{array}{rcl} n_+ & = & \#\{x | P = 0 \wedge Q > 0\} \\ n_- & = & \#\{x | P = 0 \wedge Q < 0\} \\ n_= & = & \#\{x | P = 0 \wedge Q = 0\} \end{array}$$

Then we have the following

$$\begin{array}{rcl} N(P, 1) & = & n_+ + n_- + n_= \\ N(P, Q) & = & n_+ - n_- \\ N(P^2 + Q^2, 1) & = & n_= \end{array}$$

By solving for $n_+$, we immediately obtain

$$n_+ = \frac{N(P, 1) + N(P, Q) - N(P^2 + Q^2, 1)}{2}$$

Again, when there are no free variables in the formula $(2''')$, we only need to apply Sylvester's method to see if $n_+$ is indeed $k$. When there are free variables, we need to parameterize the method. This concludes the description of Tarski's method.

### 14.2.2 Cylindrical Algebraic Decomposition Method

Collins[Coll75], a former Ph.D. student of Rosser (the logician known for Church-Rosser property of Lambda calculus) introduced a new method which has a much better bound than Tarski's (doubly exponential in the number of variables rather than non-elementary, thus the infinite tower of exponents is reduced to two floors). Specifically, the maximum computing time of the method is dominated by $(2n)^{2^{2r+8}} m^{2^{r+6}} d^3 a$, where $r$ is the number of variables in $\phi$ , $m$ is the number of polynomials occurring in $\phi$, $n$ is the maximum degree of any such polynomial in any variable, $d$ is the maximum length of any integer coefficient of any such polynomial, and $a$ is the number of occurrences of atomic formulas.

Since the introduction, this method has gone through various improvments by Arnon[Arno81], Collins[Coll91][Coll98], Hong[Hong90][Hong98][Hong90a],

McCallum[Mcca93][Mcca84] and also now has been implemented twice completely Arnon[Arno81],Hong[Hong90a]

Hong's implementation is currently used in several engineering/sicientific applications. In this section, we describe the method and various improvements.

### Method

We first begin by explaining the name of the method: *Cylindrical Algebraic Decomposition*

**Definition 1**

- A *decomposition* of $U \subseteq \mathbb{R}^r$ is a finite collection of disjoint connected subsets of $U$ whose union is $U$. Each subset is called a *cell*.
- A cell $c$ is called *algebraic* if it can be described by a quantifier-free formula. The formula is called a *defining formula* of the cell, denoted by $d_c$.
- A decomposition is called *algebraic* if every cell is algebraic
- a *stack* over $U \subseteq \mathbb{R}^r$ is a decomposition of $U \times \mathbb{R}$ such that the projection of each cell on $\mathbb{R}^r$ is exactly $U$.
- A decomposition of $D$ of $\mathbb{R}^r$ is called *cylindrical* if $r = 1$, or $r > 0$ and $D$ can be partitioned into stacks over cells of a cylindrical decomposition of $D'$ of $\mathbb{R}^{r-1}$. It is easy to see that $D'$ is unique and we call it the *induced* decomposition of $D$.
- Let $A$ be a finite subset of $\mathbb{Z}[x_1, \ldots, x_r]$. A decomposition of $\mathbb{R}'$ is called *A-sign invariant* if each polynomial in $A$ has a constant sign throughout each cell. ■

Let $F \equiv (Q_{f+1}x_{f+1}) \cdots (Q_r x_r) F(x_1, \ldots, x_r)$. Let $A$ be the set of polynomials occurring in $F$. Let $D_r$ be a $A$-sign invariant algebraic decomposition of $\mathbb{R}^r$. Let $D_k$ be the induced cylindrical algebraic decomposition of $D_{k+1}$ for $k = 1, \ldots, r-1$. We can immediately observe the following:

**Proposition 1**

- The truth of $(Q_{f+1}x_{f+1}) \cdots (Q_r x_r) F(x_1, \ldots, x_r)$ is constant throughout each cell $c$ of $D_k$. Let $v_c$ denote this truth value.
- Let $c$ be a cell of $D_r$, and let $s_c$ be a point in $c$. Then $v_c = F(s_c)$. We will call $s_c$ a *sample* point of $c$.
- Let $c$ be a cell of $D_k$, $f \leq k < r$, and let $c_1, \ldots, c_n$ be the cells in the stack over $c$. Then we have
$$v_c = \begin{cases} \vee_{i=1}^n v_{c_i} & \text{if } Q_{k+1} = \exists \\ \wedge_{i=1}^n v_{c_i} & \text{if } Q_{k+1} = \forall \end{cases}$$
- $F \Longleftrightarrow \bigvee_{c \in D_f, v_c = \text{true}} d_c$ ■

Now assuming we have an algorithm (CAD) that constructs a $A$-sign invariant cylindrical algebraic decomposition, we can immediately devise a quantifier elimination algorithm.

**Algorithm 1**

---

$$F' \longleftarrow QE(F)$$

*Input*: $F$ is a formula
*Output*: $F'$ is a quantifier-free formula equivalent to $F$

(1) $D_r \longleftarrow CAD(A)$
(2) For every cell $c$ in $D_r$, $v_c \longleftarrow F(s_c)$

(3) For $k = r-1, \ldots, f$ and for every cell $c$ in $D_k$
  (i) Let $c_1, \ldots, c_n$ be the cells in the stack over $c$
  (ii) $v_c \longleftarrow \begin{cases} \vee_{i=1}^{n} v_{c_i} & \text{if } Q_{k+1} = \exists \\ \wedge_{i=1}^{n} v_{c_i} & \text{if } Q_{k+1} = \forall \end{cases}$
(4) $F' \longleftarrow \bigvee_{c \in D_f, v_c = \text{true}} d_c$ ■

Now it remains to describe how to construct an $A$-sign invariant cylindrical algebraic decomposition. This will be done inductively on the number of variables $r$.

**CAD for** $r = 1$. Let $\alpha_1 < \cdots < \alpha_\ell$ be the real roots of the polynomials in $A$. Then the $2\ell + 1$ cells

$$(-\infty, \alpha_1), [\alpha_1, \alpha_1], (\alpha_1, \alpha_2), \ldots, (\alpha_{\ell-1}, \alpha_\ell), [\alpha_\ell, \alpha_\ell], (\alpha_\ell, \infty)$$

form a $A$-sign invariant cylindrical decomposition $D$ of $\mathbb{R}^1$. Now for each $c$ in $D$, we need to construct a sample point $s_c$ and a defining formula $d_c$. A sample point $s_c$ can be chosen to be any point of $c$.

In order to construct a defining formula $d_c$, let $\sigma_1, \ldots, \sigma_n$ be the signs of the polynomials in $A$ on $c$. Let $\rho_i$ be $>$, $=$, $<$ respectively if $\sigma_i$ is $+$, $0$, $-$. Then the formula

$$F_c \equiv A_1 \rho_1 0 \wedge \cdots \wedge A_n \rho_n 0$$

captures all points in $c$. If we are lucky, it will capture no other points, and $F_c$ can be used as a defining formula $d_c$. But it can happen that another cell shares the same signs, and $F_c$ captures that cell also.

A general way to overcome this difficulty is to "separate" the cells using derivatives.[1] Specifically, let $A'$ be the set of the derivatives of the polynomials in $A$. We inductively build a $A'$-sign invariant cylindrical algebraic decomposition $D'$ of $\mathbb{R}^1$. Then for each cell $c$ of $D$, we can easily determine, by comparing the sample points of $D'$ and $D$, the conjective cells $c'_\mu, \ldots, c'_\nu$ of $D'$ whose union contains $c$ but is disjoint from any other cell of $D$ that has the same signs as $c$. Then, we can set

$$d_c \equiv F_c \wedge \bigvee_{i=\mu}^{\nu} d_{c'_i}$$

**CAD for** $r > 1$. In order to do induction on the number of variables, it will be nice to have a finite subset $P$ of $\mathbb{Z}[x_1, \ldots, x^{r1}]$ such that we can "easily build" an $A$-sign invariant cylindrical algebraic decomposition $D$ of $\mathbb{R}^r$ from a $P$-sign invariant cylindrical algebraic decomposition $E$ of $\mathbb{R}^{r-1}$.

What do we mean by "easily build"? Let $c$ be a cell of $E$. We want to build an $A$-sign invariant stack over $c$. This becomes easy if the zeros of the polynomials in $A$ naturally "delineate" the cylinder over $c$, that is, the zeros of $A$ within the cylinder consist of disjoint graphs of continous functions from $c$ to $\mathbb{R}$, say, $f_1 < \ldots < f_\ell$. If so, the following forms an

[1] Actually, for the $r = 1$ case, one can separate the cells simply by using the sample points of the sectors. But this method is not generalizable to $r > 1$.

$A$-sign invariant stack over $c$:

$$\begin{array}{l}
\{x|x' \in c, x_r < f_1(x')\},\\
\{x|x' \in c, x_r = f_1(x')\},\\
\{x|x' \in c, f_1(x') < x_r < f_2(x')\},\\
\{x|x' \in c, x_r = f_2(x')\},\\
\{x|x' \in c, f_2(x') < x_r < f_3(x')\},\\
\vdots\\
\{x|x' \in c, x_r = f_\ell(x')\},\\
\{x|x' \in c, f_\ell(x') < x_r\}
\end{array}$$

where $x$ stands for $(x_1, \ldots, x_r)$ and $x'$ stands for $(x_1, \ldots, x_{r-1})$. Let's call them $c_1, \ldots, c_{2\ell+1}$. We call the odd-indexed cells *sectors* and the even-indexed cells *sections*. Now we need to compute sample points and defining formulas for these cells. Let $s = (s_1, \ldots, s_{r-1})$ be the sample point of $c$. Let $A^*$ be the univariate polynomials in $x_r$ obtained by evaluating the polynomials in $A$ on $s$. Then, we compute an $A^*$-sign invariant cylindrical algebraic decomposition $D^*$ of $\mathbb{R}^1$ (as shown before). Let $c_1^*, \ldots, c_{2\ell+1}^*$ be the cells of $D^*$. Let $s_1^*, \ldots, s_{2\ell+1}^*$ be their sample points. Then, we can set the following as the sample points of the cells in the stack over $c$: $(s_1, \ldots, s_{r-1}, s_i^*)$ where $1 \leq i \leq 2\ell + 1$.

Let $d_1^*, \ldots, d_{2\ell+1}^*$ be the defining formulas of the cells of $D^*$. We can obtain a defining formula for the cell $c_i$ by going through the formula $d_i^*$ and replacing every instance of the

$$\frac{d^k A_\mu^*}{dx_r^k} \text{ by } \frac{\partial^k A_\mu^*}{\partial x_r^k}$$

A caution. In order for the above argument to hold, the zeros of all orders of the partial derivative of $A$ in $x_r$ (not just merely $A$) must also delineate the cylinder over $c$. This is needed to ensure that the relative positions of the zeros of the derivatives $A^*$ do not change as $s$ ranges over $c$. If not, the defining formula for the cell $c_i^*$ might not provide a structure for the defining formula for the cell $c_i$.

Finally, it only remains to decide what to put into the set $P$ to ensure the delineability of the zeros of $A$ and its derivatives. Intuitively, we need to put sufficiently many polynomials so that their zeros contain the projection of the "critical" points of the zeros of $A$ and its derivatives. By critical points, we mean "crossing", "vertical tangent", "vertical asymptotes", and "isolated points". Collins proved that the following set $P$ is sufficient.

**Theorem 1 (Collin's projection)**

$$\begin{array}{rcl}
B & = & \{\text{red}^k(a) \mid a \in A, \deg(\text{red}^k(a)) \geq 1\}\\
L & = & \{\text{lc}(b) \mid b \in B\}\\
R & = & \{\text{psc}_k(b_1, b_2) \mid b_1, b_2 \in B, 0 \leq k < \min(\deg(b_1), \deg(b_2))\}\\
D & = & \{\text{psc}_k(b, b') \mid b \in B, 0 \leq k < \deg(b')\}\\
C & = & \{\text{der}^k(b) \mid b \in B, 0 \leq k < \deg(b)\}\\
D' & = & \{\text{psc}_k(c, c') \mid c \in C, 0 \leq k < \deg(c')\}\\
P & = & L \cup R \cup D \cup D'
\end{array}$$

where

- 'lc' stands for the leading coefficient

- 'red' for polynomial reductum, that is, red($a$) is the polynomial obtained from $a$ by removing the leading term.
- 'der' for derivative
- 'psc' for principal subresultant coefficient. Let $a = a_m x^m + \ldots + a_0$ and $b = b_n x^n + \ldots + b_0$. Then $\mathrm{psc}_k(a, b)$ is the determinant of the square (Sylvester) matrix

$$\begin{bmatrix} a_m & a_{m-1} & \cdots & a_0 & & & \\ & a_m & a_{m-1} & \cdots & a_0 & & \\ & & \cdots & \cdots & \cdots & \cdots & \\ & & & a_m & a_{m-1} & \cdots & \cdots \\ b_n & b_{n-1} & \cdots & b_0 & & & \\ & b_n & b_{n-1} & \cdots & b_0 & & \\ & & \cdots & \cdots & \cdots & \cdots & \\ & & & b_n & b_{n-1} & \cdots & \cdots \end{bmatrix}$$

  in which there are $n - k$ rows of $a$ coefficients, $m - k$ rows of $b$ coefficients, and all elements not shown are zero. ■

Roughly put, $L$ is there for vertical asymptotes, $R$ for crossing, $D$ for vertical tangents or isolated points, and $D'$ for vertical tangents/isolated points of derivatives.

Let $PROJ$ denote the map that takes $A$ and produces $P$. Summarizing, we have the following algorithm:

**Algorithm 2**

---

$$D \longleftarrow CAD(A)$$

*Input*: $A$ is a finite set of polynomials in $r$ variables
*Output*: $D$ is an $A$-sign invariant cylindrical algebraic decomposition

(1) if $r = 1$ then build a $A$-cylindrical algebraic decomposition $D$ of $\mathbb{R}^1$ as described above and return
(2) $P \longleftarrow PROJ(A)$
(3) $E \longleftarrow CAD(P)$
(4) For each cell $c$ of $E$, build an $A$-sign invariant stack $S_c$ over $c$ as described above
(5) $D \longleftarrow \cup_{c \in E} S_c$ ■

---

**Improvements**

Collins method allowed various improvements: clustering[Arno81][Mcca02], smaller projections [Mcca84] [Hong90] [Brow01a], efficient order of projections[Dolz04], partial cylindrical algebraic decomposition[Hong90a][Coll91], solution formula construction[Hong98][Brow01a], strict inequalities[Mcca93], equational constraints[Coll98], use of interval methods[Coll02], and so on. We will briefly discuss a few of the above improvements.

**Clustering** Arnon[Arno81] made an observation that a stack can be constructed over a union of cells, provided that the union (cluster) is connected and the projection polynomials are sign-invariant on it. Thus, before lifting we first combine cells into clusters and choose

only one sample point per each cluster and lift it. This usually significantly reduces the number of stack constructions which must be performed.

This idea requires a method for computing a cluster, which boils down to adjacency computation (finding out which cells are topologically adjacent). In [Arno84][Arno88b][Mcca02] methods for 2 and 3 and more variables were developed.

**Smaller Projection** In [Mcca84][Mcca88] McCallum proved that, if the polynomials in $A$ are "well" ordered, the sets $R$, $D$, $C$, $D'$ can be made smaller.

$$\begin{aligned} R &= \{\mathrm{psc}_0(a_1, a_2) \mid a_1, a_2 \in A\} \\ D &= \{\mathrm{psc}_0(a, a') \mid a \in A\} \\ C &= \{\mathrm{der}^k(a) \mid a \in A, 0 \leq k < \deg(a)\} \\ D' &= \{\mathrm{psc}_0(c, c') \mid c \in C\} \end{aligned}$$

Note that these sets are much smaller than Collins original ones, because it does not involve reductums nor higher order principal subresultant coefficients.

Hong[Hong90] proved that the set R in the original projection could be restricted to (without any side conditions)

$$R = \{\mathrm{psc}_k(a, b) \mid a \in A, b \in B, 0 \leq k < \min(\deg(a), \deg(b))\}$$

The resulting projection set is smaller than the original one and larger than McCallums. But it can be useful since it, unlike McCallums, does not impose any condition on the polynomials and further McCallums projection requires that the clusters must be order invariant. These can result in possibly smaller clusters. Brown[Brow01a] made a further improvement.

**Partial CADs** Hong[Hong90a] (see also [Coll91]) showed that we can *very often* complete quantifier elimination by a partially built cylindrical algebraic decomposition, if we utilize, during cylindrical algebraic decomposition, more information contained in the input formula such as quantifiers, the boolean connectives, the absence of some variables from some polynomials occurring in the input formula, etc. These improvements did not change the asymptotic worst case bound, but gave significant speedups in most problems known in the literature, that many problems that would require at least several months could now be solved within a few seconds.

As an example, one can utilize the quantifier information as follows: Let us consider a sentence in two variables $(\exists x)(\exists y)F(x, y)$. The original method computes a certain decomposition $D_1$ of $\mathbb{R}$ and then lifts this to a decomposition $D_2$ of $\mathbb{R}^2$ by constructing a stack of cells in the cylinder over each cell of $D_1$. Then the quantifier elimination proceeds by determining the set of all cells of $D_1$ in which $(\exists y)F(x, y)$ is true. Finally, it computes the truth value of $(\exists x)(\exists y)F(x, y)$ by checking whether the set is empty. In contrast, one may construct only one stack at a time, aborting the CAD construction as soon as a cell of $D_1$ is found which satisfies $(\exists y)F(x, y)$, if any such cell exists. The method illustrated above for two variables extends in an obvious way to more variables, with even greater effectiveness because the CAD construction can be partial in each dimension.

**Simple Solution Formula Construction**

Hong[Hong90a][Hong98] devised a new method for constructing solution formulas which is more efficient and produces much simpler formulas. The original algorithm obtains a solution formula by forming a disjunction of defining formulas of solution cells. This method works for any input formula, but produces very large formulas and often increases greatly the amount of computation required because of the augmented projection (projection involving derivatives).

The method of Hong does not use augmented projection, but instead tries to construct solution formulas using only projection polynomials. It can fail to produce a solution formula, but the experiments with many QE problems from diverse application areas suggest that it will rarely fail. The method also uses a logic minimization algorithm to simplify solution formulas. It carries out simplification based not only on the logical connectives but also on the relational operators. This is done with three-valued logic. It further reduces the size of the inputs to the multiple-valued logic minimization algorithm by taking advantage of the structure of the input formula. As the result, the method produces simpler formulas (a few lines instead of several pages) faster (a few seconds instead of hours).

Brown[Brow99] provided a method that generates the augmented projection polynomials on demand, thus making the method complete (never fails, unlike Hongs).

### Strict Equalities

McCallum[Mcca93] observed that the stack construction phase can be significantly improved if the input formula is an existentially quantified sentence of a system of strict inequalities. This kind of sentence arises naturally from geometric modeling, robot simulation, non-linear optimization, etc.

The key observation is that the solution set of the quantifier-free matrix (conjunction/disjunction of strict inequalities) is either empty or must contain a full-dimension open set. Thus, during the stack construction, one only needs build stacks over sectors (since any cells belonging to stacks over sections cannot not be of full-dimension). If one of the constructed full-dimensional cells satisfies the matrix, then the sentence is true. If not, one can stop and report that the sentence is false.

This gives a huge savings not only because the number of the stack constructions is reduced, but also the full-dimensional cells are much cheaper to construct since it does not involve algebraic number computation.

### Equational Constraints

McCallum[Mcca93] observed that the projection phase and the stack construction phase can be significantly improved when the input formula has "equational constraints". A *equational constraint* of a prenex formula is a polynomial equation which is implied by its quantifier-free matrix. For example, if the quantifier-free matrix is of the form: $A = 0 \wedge F$, then $A = 0$ is an equational constraint.

The key observation is that the matrix is false if $A \neq 0$ regardless of the truth of $F$. Thus, we only need to ensure that the other polynomials (occurring in $F$) are sign-invariant on the sections of $A$. For a simple presentation, let us assume that the inputs polynomials are well-oriented (and thus McCallums projection theorem can be applied). Then, we only need to put the following into the projection set:

- The discriminant and (enough of) the coefficients of $A$

- The resultant of $A$ and each polynomial occuring in $F$

Sometimes we can propagate the property of "equational constraint" down to lower dimensional space so that we can apply the smaller projection operation again. For instance, let $A_1$ and $A_2$ both be equational constraint polynomials. Let $R$ be their resultant. It is well-known that $A_1 = 0 \wedge A_2 = 0 \Longrightarrow R = 0$ Thus, $R = 0$ is also an equational constraint, we can use it to reduce the second projection. This idea obviously generalizes, with greater effect, when there are more than two equational constraints.

When there are equational constrains obtained by resultant computations (as shown above), then they can be used for pruning stack constructions. Whenever a stack has already been constructed in which there are sections of an equational constraint polynomial, all other cells in the stack can be marked false and it is unnecessary to construct stacks over them. When there are many equational constraints in the input formulas, the accumulated reduction on the number of stack constructions can be drastic.

### 14.2.3 Quantifier-Block Elimination Methods

During last decade, there was an intensive research on improving asymptotic worst case computing bounds [Beno86], [Fitc87], [Grig88], [Grig88a], [Cann88], [Cann93], [Hein89], [Rene92], [Weis98]. The key idea is that a consecutive block of same quantifiers can be eliminated by a single projection in an exponential time (in the number of variables). Thus, the total complexity is doubly exponential in the number of quantifier alternations, not on the number of variables. If all quantifiers are the same (1 quantifier-block), then the total complexity is singly exponential in the number of variables.

Let $n$ be the number of variables in the input sentence, $m$ the number of polynomials, $d$ the degree, $L$ the bit length bound for the coefficients, and $n_i$ is the number of quantifiers in the $i$-th quantifier block. The best asymptotic bound so far (obtained by Renegar) is is $L(log\ L)(log\ log\ L)(md)^{\prod_{k=0}^{\omega} O(n_k)}$.

#### Method

There are several methods with the similar complexity bounds. [Grig88a], [Cann88], [Cann93], [Hein89], [Rene92]. Though different in details, they are very similar in the overall strategy. Thus in this tutorial, we will be satisfied with sketching the overall strategy/ideas.

#### Overall Ideas

1. First note that the arbitrary quantifier elimination problem can be trivially reduced to the *existential* quantifier elimination problem (all quantifiers are existential), because if we have an algorithm for the existential quantifier elimination, we can repeatedly apply it to eliminate each block of quantifiers one at a time (for a universal block after double negation).

   One important exception. Renegar's method does not apply the existential quantifier elimination repeatedly. Instead, it follows the idea of Collins' Cylindrical Algebraic Decomposition: namely the repeated projection. But its projection removes a block of variables at once, while Collins' projection removes one variable one at time.

2. The existential quantifier elimination can be straightforwardly (though very tedious) reduced to the *existential* decision problem (no free variables), because we can "un-

ravel" a decision algorithm to turn it into a parametric one. More specifically, a decision algorithm can be viewed as a tree where each internal node corresponds computation (such addition/multiplication) or comparison (for branching) and where each leaf is associated with true or false. Given a sentence, we follow only one path until we reach a leaf. If it is a true node, then the sentence is true, else false. But when the input is not a sentence and contains free variables, we do not know which path to follow at a branching node. Thus, we follow all the paths while accumulating the branch conditions (on the free variables) associated with each path. Then, the disjunction of all the accumulated conditions associated with true leaves will be the quantifier-free formula that we desire.

Again, Renegar does not carry this unraveling at each quantifier block. But he does this only once for the "last" free variable block.

3. The existential decision problem can be reduced to the *smooth* existential decision problem where the solution set of the quantifier-free matrix is closed and smooth. This can be done by some simple geometric tricks and (careful) infinitesimal smoothing.

4. Next the smooth existential decision problem is reduced to the *equationally constrained* existential decision problem:

$$\exists(x_1, \ldots, x_n) P_1(x_1, \ldots, x_n) \wedge \cdots P_n(x_1, \ldots, x_n) \wedge F$$

where $F$ is a quantifier-free formula and the system $P_1 = P_2 = \cdots = P_n = 0$ has only finitely many (complex) solutions. This can be done by applying the Lagrange multiplier method since a smooth set has a "optimal/critical" point for "suitably" chosen objective functions (Morse function). The suitable choice can be done either by (another) infinitesimal perturbation or some searching (in a finite but very big set).

Note that this idea of using "optimal/critical" points is not new, but was already used by Seidenberg[Seid54] for improving Tarski's method. He used the "distance" from the origin as the objective function. When this turns out to be not suitable, then he tried linear transformation of coordinates.

Weispfenning's method is an exception here in that it does not require that the system $P_1, \ldots, P_n$ has a finitely many complex solution. It checks if the system indeed has the property, if so continue, continue, if not, instead of perturbing the system, it view some of the variables as parameters and checks again whether the system has finitely many complex solutions (parametrically). This can be done by using the method of comprehensive Gröbner basis method [Weis92]. Thus in the worst case, the complexity of this method becomes doubly exponential in the number of variables. But some preliminary experiments show that this might be a very promising method for small inputs.

5. The equationally constrained existential decision problem can be solved in various ways. Usually, the system of equations are "solved" by using the $u$-resultants [Cann87] or the Hermite's quadratic form [Pede93] The sign computation of the polynomials occurring $F$ on the roots of the system of equations can be obtained by the method of BKR [Beno86].

**Grigorev's Algorithm**: Now for those who want to get a glance at the details of some algorithms, we give a detailed description of the algorithm of Grigor'ev and Vorobjov [Grig88] and Renegar's [Rene92], but without explanation. First we give Grigorev's algorithm for existential decision problem. In the case of a positive answer, the algorithm also constructs

a representative set for the family of components of connectivity of the set of all real solutions of the system, which is $\mathcal{T}$ in Step (8) of the algorithm described below. We begin by defining several notations which will be used in the algorithm description:

- Let $K$ be an arbitrary ordered field. The $\tilde{K}$ denotes the algebraic closure $K$, and $\tilde{K}$ the real algebraic closure of $K$.
- Let $\epsilon_1$ be a positive infinitesimal over $\tilde{\mathbb{Q}}$. Then $F_1$ denotes the real algebraic closure of $\tilde{\mathbb{Q}}(\epsilon_1)$.
- Let $\epsilon$ be a positive infinitesimal over $F_1$. Then $F$ denotes the real algebraic closure of $F_1(\epsilon)$.
- Let $f_1, \ldots, f_m$ be elements of $K[x_1, \ldots, x_n]$. Then $\{f_1 = 0, \ldots, f_m = 0\}$ denotes the set of all solutions of the system $f_1 = \cdots = f_m = 0$ over the algebraic closure of $K$.

---

$$t \longleftarrow \textbf{Decide}(P)$$

*Input*: $P$ is a quantifier-free formula of the following kind:

$$f_1 > 0 \wedge \cdots \wedge f_k > 0 \wedge f_{k+1} \geq 0 \wedge \cdots \wedge f_m \geq 0$$

where $f_i \in \mathbb{Z}[x_1, \ldots, x_n]$.

*Output*: $t$ is the truth of the sentence $(\exists x_1 \in \mathbb{R}) \cdots (\exists x_n \in \mathbb{R}) P(x_1, \ldots, x_n)$

(1) Let $f_{m+1} = x_0 f_1 \cdots f_{k-1}$
Let $g_1 = (f_1 + \epsilon_1) \cdots (f_{m+1} + \epsilon_1) - \epsilon_1^{m+1} \in \mathbb{Z}[\epsilon_1][x_0, \ldots, x_n]$
(2) Let $\tilde{L}$ be a bit length bound for the coefficients of $f_i$, and $\tilde{d}$ a degree bound for $f_i$ for every $1 \leq i \leq m+1$.
Let $R = 3^{(\tilde{L}+log(m+1))p(\tilde{d}^{n+1})}$, where $p \in \mathbb{Z}[x]$ is a certain polynomial defined in Page 62 of [Grig88].
Let $g = g_1^2 + (x_0^2 + \cdots + x_{n+1}^2 - (R+1))^2 \in \mathbb{Z}[\epsilon_1][x_0, \ldots, x_{n+1}]$
(3) Let $N = (8md)^{n+2}$
Let $\Gamma = \{1, \ldots, N\}^{n+1}$
Let $\mathcal{I}' = \{\}$
(4) For each $\gamma = (\gamma_1, \ldots, \gamma_{n+1}) \in \Gamma$ do
(4.1) Let $\tilde{V}^{(\epsilon)} \subset \tilde{F}^{n+2}$ be the variety of the system

$$\begin{aligned} g - \epsilon &= \left(\frac{\partial g}{\partial x_1}\right)^2 - \frac{\gamma_1}{N(n+2)}\Delta = \cdots \\ &= \left(\frac{\partial g}{\partial x_{n+1}}\right)^2 - \frac{\gamma_{n+1}}{N(n+2)}\Delta = 0 \end{aligned}$$

where $\Delta = \sum_{j=0}^{n+1} \left(\frac{\partial g}{\partial x_j}\right)^2$
Let $\tilde{V}^{(\epsilon)} = \bigcup_j \tilde{V}_j^{(\epsilon)}$, where each $\tilde{V}_j^{(\epsilon)}$ is a component irreducible over the field $\mathbb{Q}(\epsilon_1, \epsilon)$.
(4.2) For each null dimensional $\tilde{V}_j^{(\epsilon)}$ do
(4.2.1) Let $\mathcal{R} \subset \tilde{F}_1^n$ be a set containing the standard part (relative to $\epsilon$) of every point (for which the standard part is definable) from the component $\tilde{V}_j^{(\epsilon)}$.

(4.2.2) Let $\mathcal{R}'_1 = \mathcal{R}_1 \cap \{g = 0\} \cap F_1^{n+2}$
(4.2.3) Set $\mathcal{I}' to\ \mathcal{I}' \cup \mathcal{R}'_1$
(5) Let $\mathcal{I} = \pi(\mathcal{I}') \subset F_1^{n+1}$ where $\pi(x_0, \ldots, x_{n+1}) = (x_0, \ldots, x_n)$.
(6) Let $\mathcal{R} \subset \tilde{\mathbb{Q}}^{n+1}$ be a set containing the standard part (relative to $\epsilon_1$) of every point (for which the standard part is definable) from the set $\mathcal{I}$.
(7) Let $\mathcal{T}' = \mathcal{R} \cap \{f_1 \geq 0, \ldots, f_{m+1} \geq 0\} \cap \tilde{\mathbb{Q}}^{n+1}$
(8) Let $\mathcal{T} = \pi_1(\mathcal{T}')$ where $\pi_1(x_0, \ldots, x_n) = (x_1, \ldots, x_n)$
(9) Finally let $t$ be true if $\mathcal{T}$ is non-empty, false otherwise. ■

---

In [Grig88] it is shown that the worst case time complexity of this algorithm is dominated by $L(md)^{n^2}$.

In [Grig88a] Grigor'ev, by generalizing this algorithm, gave a decision algorithm for the first order theory of real closed fields.

**Renegar's Method**. Now we give an overview of Renegar's[Rene92] algorithm.

---

$t \longleftarrow$ **MainAlgorithm**$(P)$

*Input*: $P$ is a quantifier-free formula with variables $x_1, \ldots, x_n$.
*Output*: $t$ is the truth of the sentence $(\exists x_1 \in \mathbb{R}) \cdots (\exists x_n \in \mathbb{R}) P(x_1, \ldots, x_n)$.

(1) Let $g = \{g_1, \ldots, g_m\}$,$g_i \in \mathbb{Z}[x_1, \ldots, x_n]$ be the polynomials occurring in the formula $P$.
Let $h = \{h_1, \ldots, h_{6m+2}\}$ where

$$\begin{array}{lcll} h_i & = & g_i & i = 1, \ldots, m \\ h_{m+i} & = & x_0 g_i - 1 & i = 1, \ldots, m \\ h_{2m+i} & = & x_0 g_i + 1 & i = 1, \ldots, m \\ h_{3m+1} & = & x_0 - 1 & \\ h_{3m+1+i} & = & -g_i & i = 1, \ldots, m \\ h_{4m+1+i} & = & -x_0 g_i + 1 & i = 1, \ldots, m \\ h_{5m+1+i} & = & -x_0 g_i - 1 & i = 1, \ldots, m \\ h_{6m+2} & = & -x_0 + 1 & \end{array}$$

(2) Let $d$ be the maximum of the degrees of the polynomials $g_i$
Let $d'$ be the least even integer which is greater than or equal to $d + 1$.
Let $\tilde{h} = \{\tilde{h}_1, \ldots, \tilde{h}_{6m+2}\}$ where

$$\tilde{h}_i(\delta; x_0, \ldots, x_n) = (1 - \delta) h_i + \delta(1 + \sum_{j=0}^{n} i^j x_j^{d'})$$

(3) For each $A \subseteq \{1, \ldots, 6m + 2\}$ such that $|A| \leq n + 1$ do:
(3.1) Let $\tilde{h}(x_0, \ldots, x_n) = \sum_{j=0}^{n} (6m + 3)^j x_j^{d'}$
Let $M_A$ be the matrix with the last row $\nabla_{x_0,\ldots,x_n} \hat{h}$ and with earlier rows $\nabla_{x_0,\ldots,x_n} \overline{h}$, $i \in A$, ordered by increasing indices $i$.
(3.2) Let $h_A(\delta; x_0, \ldots, x_n) = \det(M_A M_A^T) + \sum_{i \in A} \overline{h}_i^2$
Let $d_A$ be the degree of $h_A$ with respect to $x_0, \ldots, x_n$
(3.3) Let $\tilde{h}_A(\epsilon, \delta; x_0, \ldots, x_n) = (1 - \epsilon) h_A - \epsilon \sum_{j=0}^{n} x_j^{d_A}$
Let $\tilde{h}_A^{(i)} = \frac{\partial \tilde{h}_A}{\partial x_i}$ for each $i$

(3.4) Let $R_A(\epsilon, \delta; u_0, \ldots, u_{n+1})$ be the $u$-resultant of the polynomials $\tilde{h}_A^{(0)}, \ldots, \tilde{h}_A^{(n)}$ (Use the subalgorithm shown below)

(3.5) Let $R_A$ be expanded as $\sum_{i,j,k} \epsilon^i \delta^j u_0^k R_A^{<i,j,k>}$

(4) Let $\mathcal{R}$ be the set of all $\mathcal{R}_A^{<i,j,k>}$ computed in Step (3)

(5) Let $\mathcal{B}_{n+1,D} = \{(i^{n-1}, i^{n-2}, \ldots, 1, 0) | 0 \leq i \leq nD^2\}$
For $R \in \mathcal{R}$, let $D_R$ be the degree of $R$
Let $\mathcal{Q} = \{\frac{d^j}{dt^j} \nabla_{u_1,\ldots,u_{n+1}} R(\beta + te_{n+1}) | R \in \mathcal{R}, \beta \in \mathcal{B}_{n+1,D_R}, j \in [0, D_R]\}$

(6) Let $G = \{G_1, \ldots, G_m\}$, $G_i \in \mathbb{Z}[x_1, \ldots, x_{n+1}]$ be such that each $G_i$ is the degree $d$-homogenization of $g_i$, i.e. the monomials of $G_i$ are obtained from those of $g_i$ by multiplying by the appropriate powers of $x_{n+1}$ so as to become of degree $d$
Let $\mathcal{F}^+ = \{(G_1(q), \ldots, G_m(q), q_{n+1} | q \in \mathcal{Q}\}$
Let $\mathcal{F}^- = \{(G_1(-q), \ldots, G_m(-q), -q_{n+1} | q \in \mathcal{Q}\}$
Let $\mathcal{F} = \mathcal{F}^+ \cup \mathcal{F}^-$

(7) For each $f \in \mathcal{F}$ let $\overline{S}_f$ be the set of all consistent sign vectors for the polynomials in $f$. (Use the algorithm of Ben-Or, Kozen, and Reif[Beno86])

(8) Let $\overline{S} = \bigcup_{f \in \mathcal{F}} \overline{S}_f$
Let $S = \{(\sigma_1, \ldots, \sigma_m) | (\sigma_1, \ldots, \sigma_m, 1) \in \overline{S}\}$
Finally let $t = \bigvee_{\sigma \in S} P_\sigma$, where the formula $P_\sigma$ is obtained from the formula $P$ by replacing $g_i$ with $\sigma_i$ ■

*Input*: $f_0, \ldots, f_n \in W[x_0, \ldots, x_n]$, where $W$ is a commutatuve ring with unity.
*Output*: $R \in W[u_0, \ldots, u_{n+1}]$ is the $u$-resultant of $f_0, \ldots, f_n$.
(See Renegar[Rene92, Rene92a, Rene92b] for the definition)

(1) Let $\tilde{d}$ be the maximum of the degrees of $f_i$
Let $\mathbb{B} = \{x_0^{d_0} \cdots x_{n+1}^{d_{n+1}} | d_0 + \cdots + d_{n+1} = \hat{d}\}$ where $\hat{d} = (n+1)(\tilde{d} - 1) + 1$
Let $\mathbb{H}$ be the vector space generated by the basis $\mathbb{B}$ over the ring $W$

(2) For each $x_0^{d_0} \cdots x_{n+1}^{d_{n+1}} \in \mathbb{B}$, let $i$ denote the least index $i \leq n$ and $\tilde{d} \leq d_i$, if such an $i$ exists, otherwise let $i = n + 1$
Let $F_i \in W[x_0, \ldots, x_{n+1}]$ be the degreee $\tilde{d}$-homogenization of $f_i$ for each $i$
Let $\mathcal{T} : \mathbb{H} \longrightarrow \mathbb{H}$ be the linear transformation defined by the following mapping on the basis $\mathbb{B}$

$$\mathcal{T}(x_0^{d_0} \cdots x_{n+1}^{d_{n+1}}) = \begin{cases} x_0^{d_0} \cdots x_i^{d_i - \tilde{d}} \cdots x_{n+1}^{d_{n+1}} F_i & \text{if } i \leq n \\ x_0^{d_0} \cdots x_n^{d_n} x_{n+1}^{d_{n+1}-1} u \cdot x & \text{if } i = n+1 \end{cases}$$

where $u = (u_0, \ldots, u_{n+1})$ and $x = (x_0, \ldots, x_{n+1})$
Let $M$ be the matrix representing $\mathcal{T}$ with respect to the basis $\mathbb{B}$

(3) Finally set $R = D! \det(M)$ where $m$ is a $D \times D$ matrix.■

In [Rene92] it is shown that the worst case time complexity of this algorithm is dominated by $L(log\ L)(log\ log\ L)(md)^{O(n)}$, which is the best compared to the theoretical complexities of all other algorithms proposed in the literature so far.

Renegar[Rene92], by generalizing the above algorithm, gave a quantifier elimination algorithm for the first order theory of the reals.

## 14.3 Special Methods

During last several years, there have been efforts for developing methods that work on a restricted class of inputs. Such research is motivated by the observation that one can identify interesting and useful sub-class of problems and that one might be able to develop more efficient (in practice) methods for them.

### 14.3.1 Low Degrees

Weispfenning[Weis88][Weis94][Weis97], Loos[Loos93], Anai[Anai00] and others devised methods for formulas where the bound variables occur in low degrees (1,2, and 3).

#### Restricted Problem Class

A formula with one quantifier is called *degree* $n$ if the bound variable occur in degree at most $n$. We would like to find methods for formulas with small degrees.

For formulas with many quantifiers, we can repeatedly apply such a method to eliminate all the quantifiers one at a time, starting from the innermost ones, provided that the result of each quantifier elimination stays degree at most $n$ (in the remaining bound variables).

#### Method

In order to get the main intuition, let us consider the case when there are no free variables. The initial idea is similar to that of the Cylindrical Algebraic Decomposition.

Let $\alpha_1 < \cdots < \alpha_\ell$ be the real roots of the polynomials occurring in the quantifier-free matrix $F$. Then the sign of each polynomial (and thus the truth of the sentence) is constant throughout each cell:

$$(-\infty, \alpha_1), [\alpha_1, \alpha_1], (\alpha_1, \alpha_2), \ldots, (\alpha_{\ell-1}), [\alpha_\ell, \alpha_\ell], (\alpha_\ell, \infty)$$

Thus, we only need to "sample" one point from each cell. For instance, we could choose

$$S = \{\alpha_i | 1 \leq i \leq \ell\} \cup \{\alpha_\ell + 1, \alpha_1 - 1\} \cup \{\frac{1}{2}(\alpha_i + \alpha_{i+1}) | 1 \leq i \leq \ell\}$$

Then we have

$$\exists x F(x) \Longleftrightarrow \bigvee_{t \in S} F(t)$$

$$\forall x F(x) \Longleftrightarrow \bigwedge_{t \in S} F(t)$$

Thus, the finite set $S$ contains sufficiently many samples from an infinite set $\mathbb{R}$ for the purpose of deciding the sentences. Let us call such a set the *sample set*.

This method is generalizable in a obvious way to the case with free variables, but not *efficiently*, because when there are free variables, the order of the roots depend on the values of the free variables and we will have anticipate all the "potential" orders, blowing up the size of the output formula.

A better approach (taken by Weispfenning) is to utilize the symbolic devices such as infinities and (positive) infinitesimals: $\infty$ and $\epsilon$. Using these, one can see immediately the following

set also forms a sample set (though a rigorous proof will require either the tedious $\epsilon - \delta$ reasoning or the non-standard analysis):

$$S = \{\alpha_i \mid 1 \leq i \leq \ell\} \cup \{-\infty\} \cup \{\alpha_i + \epsilon \mid 1 \leq i \leq \ell\}$$

Note that the resulting formula is quantifier-free but it instead contains other new symbols such as $\infty$, and the "roots" $\alpha_i$. Now, we need to eliminate them, thus the quantifier elimination problem is reduced to infinity elimination, infinitesimal elimination, and root elimination. Clearly it suffices to show how to eliminate those symbols from each atomic formula.

**Infinity Elimination**: Let $f_n(x) = a_n x^n + \cdots + a_0$ be a polynomial in an atomic formula. Then from the intended meaning of $-\infty$, the correctness of the following rewrite rules is immediate:

$$\begin{array}{lcl} f_n(-\infty) = 0 & \rightarrow & a_n = 0 \wedge \cdots \wedge a_0 = 0, \\ f_n(-\infty) < 0 & \rightarrow & (-1)^n a_n < 0 \vee [a_n = 0 \wedge f_{n-1}(-\infty) < 0] \end{array}$$

For the case of $<$, we only need to apply the rule repeatedly until $n$ becomes 1. Atomic formulas with other relational operators can be rewritten to formulas involving only $=$ and $<$ to which the above rules can be applied.

**Infinitesimal Elimination**: Let $f(x) = a_n x^n + \cdots + a_0$ be a polynomial in an atomic formula. From the intended meaning of $\epsilon$, we have the followings (assuming that $f$ is a non-zero polynomial):

- $f(\alpha + \epsilon) \neq 0$
- The sign of $f(\alpha + \epsilon)$ is the same as that of the highest non-vanishing derivative of $f$ an $\alpha$.

From these observations, we immediately obtain the following rewrite rules:

$$\begin{array}{lcl} f(\alpha + \epsilon) = 0 & \rightarrow & a_n = 0 \wedge \cdots \wedge a_0 = 0 \\ f(\alpha + \epsilon) < 0 & \rightarrow & f(\alpha) < 0 \vee [f(\alpha) = 0 \wedge f'(\alpha + \epsilon) < 0] \end{array}$$

For the case of $<$, we only need to apply the rule repeatedly until the degree of $f$ becomes 1.

**Linear Root Elimination**: Let $f$ be a polynomial in $x$. Let $\alpha$ be a real root of a linear polynomial $a_1 x + a_0 = 0$, thus $\alpha = -a_0/a_1$. We would like to eliminate the root (or root term) from the atomic formula $f(-a_0/a_1)\ \rho\ 0$ where $\rho$ is a relational operator. Essentially we would like to eliminate the division symbol: /. One could immediately think of the following rewrite rule:

$$\begin{array}{lcl} f(b/a) = 0 & \rightarrow & a^n f(b/a) = 0 \\ f(b/a) < 0 & \rightarrow & a^{n+\delta} f(b/a) < 0 \end{array}$$

where $n$ is the formal degree of $f$, $\delta$ is its parity, and $a^k f(b/a)$ stands for the polynomial obtained by canceling out the denominators of $f(b/a)$ by multiplying with $a^k$. But this is not correct. The denominator $a$ might vanish for some values of the free variables, thus making the root undefined. But it also means that it is not a root for the specific values of the free variables. Thus, we could simply ignore it. But it can happen that all roots are such. In

that case, we should not ignore all of them, since we need to have at least one sample point. One solution is to view such a case (vanishing denominator) as always supplying the sample point 0. The following rewrite rules implement these ideas.[2]

$$\begin{array}{lcl} f(b/a) = 0 & \rightarrow & [a = 0 \wedge f(0) = 0] \vee [a \neq 0 \wedge b^n f(b/a) = 0] \\ f(b/a) < 0 & \rightarrow & [a = 0 \wedge f(0) < 0] \vee [a \neq 0 \wedge b^{n+\delta} f(b/a) < 0] \end{array}$$

**Quadratic Root Elimination**: Let $f$ be a polynomial in $x$. Let $\alpha$ be a real root of a quadratic polynomial $a_2x^2 + a_1x + a_0 = 0$, thus $\alpha$ is one of $\frac{-a_1 \pm \sqrt{a_1^2 - a_2a_0}}{2a_2}$. We would like to eliminate the root from the atomic formula $f(\frac{-a_1 \pm \sqrt{a_1^2 - a_2a_0}}{2a_2})\ \rho\ 0$ where $\rho$ is a relational operator. Essentially we would like to eliminate the symbols: $\sqrt{\ }$ and $/$. The following observation helps.

$$\frac{a + b\sqrt{c}}{d} + \frac{a' + b'\sqrt{c'}}{d'} = \frac{(ad' + a'd) + (bd' + b'd)\sqrt{c}}{dd'}$$

$$\frac{a + b\sqrt{c}}{d} \times \frac{a' + b'\sqrt{c'}}{d'} = \frac{(aa' + bb'c) + (ab' + a'b)\sqrt{c}}{dd'}$$

By applying these equality repeatedly, $f(\frac{a+b\sqrt{c}}{d})$ results in the same form:

$$\frac{a^* + b^*\sqrt{c^*}}{d^*}$$

Note that $d^* = d^k$ where $k$ is the formal degree of $f$. Let $\delta$ be the parity of $k$. Then we can rewrite:

$$f(\frac{a + b\sqrt{c}}{d}) = 0 \quad \rightarrow \quad a^{*^2} - b^{*^2}c = 0 \wedge a^*b^* \leq 0$$

$$f(\frac{a + b\sqrt{c}}{d}) < 0 \quad \rightarrow \quad [[a^*d^\delta \leq 0 \wedge a^{*^2} - b^{*^2}c \geq 0] \vee b^*d^\delta \leq 0] \wedge$$

$$[a^*d^\delta \leq 0 \vee a^{*^2} - b^{*^2}c \leq 0]$$

Again we need to take care of the "bad" situations such as $d = 0$ or $c < 0$. This can be done by doing some case analysis, and this will increase the size of the formula a bit.

**Improvements** In [Loos93], several optimizations are described, such as reducing the size of the sample set by utilizing the relational operators, the boolean structure, the quantification structure, etc.

---

[2] At the time of writing this tutorial, I notice that it is not actually necessary to duplicate 0 so many times. We need only one 0 to ensure that there is at least one sample point. Further $a \neq 0$ can be "factored out". So we can go back to the "incorrect" rewrite rules:

$$\begin{array}{lcl} f(b/a) = 0 & \rightarrow & a^n f(b/a) = 0 \\ f(b/a) < 0 & \rightarrow & a^{n+\delta} f(b/a) < 0 \end{array}$$

provided that we use

$$\exists x F \Longleftrightarrow F(0) \vee \bigvee_{b/a \in S} a \neq 0 \wedge F(b/a)$$

### 14.3.2 Constrained by Quadratic Equation

Hong[Hong93][Hong93a] devised an algorithm that eliminates a quantifier from a formula which is *constrained by a quadratic equation*. The output formulas are made of resultants and their variants called *slope* resultants. The slope resultants can be, like the resultants, expressed as determinants of certain matrices.

Weispfenning[Weis94] gave a method (as described in the previous section) which handles this case (and more). The method there does not require extended resultant calculus. The outputs are also very similar except that the method of Weispfenning systematically introduces some extraneous factors in the output polynomials. In fact, the initial motivation for developing the variant resultant calculus was to avoid the introduction of extraneous factors. Further the variant resultant calculus has some nice properties and might provide some new way to higher degrees.

#### Restricted Problem Class

A formula is said to be *constrained by a quadratic equation* if it is of the following form:

$$(\exists x \in \mathbb{R})[a_2x^2 + a_1x + a_0 = 0 \wedge F]$$

where

- $F$ is a quantifier free formula in $x_1, \ldots, x_r, x$
- $a_2, a_1, a_0$ are polynomials over $x_1, \ldots, x_r$ such that $a_2, a_1$ and $a_0$ do not have a common real zero in $\mathbb{R}^r$.

#### Mathematical Tool: Slope Resultants

Let us first define a variant of resultant, which we call *slope* resultant. Later we will use this while developing a quantifier elimination algorithm for the problem stated above.

Let $I$ be an integral domain, and let $\tilde{I}$ be the unique (up to isomorphism) algebraic closure of the quotient field of $I$.

**Defintion 2 (Slope)** *Let $P$ be a univariate polynomial over $I$. The $k$-th slope of $P$, written as $P^{<k>}$, for $k \geq 0$, is the $(k+1)$-variate polynomial over $I$ defined recursively by*

$$\begin{aligned} P^{<0>}(x_1) &= P(x_1) \\ P^{<k>}(x_1, \ldots, x_{k+1}) &= \frac{P^{<k-1>}(x_1, \ldots, x_k) - P^{<k-1>}(x_2, \ldots, x_{k+1})}{x_1 - x_{k+1}} \end{aligned}$$

*for $k \geq 1$.*■

Though the definition of slope involves rational functions, the divisions are always exact, and thus a slope is a polynomial. We keep the rational function formulation in this definition because it is more natural and intuitive.

**Definition 3 (Slope Resultant)** *Let $A$ and $B$ be univariate polynomials over $I$. Let $A = a_m \prod_{i=1}^{m}(x - \alpha_i)$ where $\alpha_i \in \tilde{I}$. Then the $k$-th slope resultant of $A$ and $B$, written*

*as* $\mathrm{sres}_k(A,B)$*, is defined by*

$$\mathrm{sres}_k(A,B) = a_m^{n-k} \sum_{1\le i_1<\cdots<i_{k+1}\le m} B^{<k>}(\alpha_{i_1},\ldots\alpha_{i_{k+1}}) \quad \blacksquare$$

Intuitively this is the average of the $k$-th "derivative" of $B$ at the roots of $A$, up to a constant factor.

The slope resultant can be computed in various different ways: recurrence formula, generating functions, and determinants of certain matrix [Hong98a][Hong93a]. Here we only give the determinant method.

**Theorem 2 (Slope Resultant as Determinant)** *Let* $A = \sum_{i=0}^{m} a_i x^i$ *and* $B = \sum_{i=0}^{n} b_i x^i$*. Then we have*

$$\mathrm{sres}_k(A,B) = \det(M)$$

*where* $M$ *is the* $n+1-k$ *by* $n+1-k$ *matrix defined by*

$$M = \begin{bmatrix} a_m & a_{m-1} & \cdots & \cdots & \cdots & \\ & \cdots & \cdots & \cdots & \cdots & \cdots \\ & & a_m & a_{m-1} & a_{m-2} & c_3^{k+1}\cdot a_{m-3} \\ & & & a_m & a_{m-1} & c_2^{k+1}\cdot a_{m-2} \\ & & & & a_m & c_1^{k+1}\cdot a_{m-1} \\ b_n & b_{n-1} & \cdots & \cdots & b_{k+1} & c_m^{k+1}\cdot b_k \end{bmatrix}$$

*where* $c_i^k = \binom{m}{k} - \binom{m-i}{k}$*. Let* $n' = n+1-k$*. Precisely, the matrix is defined by*

$$M_{i,j} = \begin{cases} a_{m-(j-i)} & \text{if } i < n' \text{ and } j < n' \\ b_{n-j+1} & \text{if } i = n' \text{ and } j < n' \\ c_{j-i}^{k+1} a_{m-(j-i)} & \text{if } i < n' \text{ and } j = n' \\ c_m^{k+1} b_k & \text{if } i = n' \text{ and } j = n' \end{cases}$$

*where* $a_\mu = 0$ *if* $\mu > m$ *or* $\mu < 0$ *and* $b_\nu = 0$ *if* $\nu > n$ *or* $\nu < 0$ $\blacksquare$

### Method

Now we are ready to give a quantifier elimination algorithm that utilizes the slope resultants. Without losing generality, let us assume that the quantifier free forula $F$ involves only the two relation operators: $=$ and $>$. We can decompose the input formula $F^*$ into two sub-problems in an obvious way.

$$F^* \Longleftrightarrow [a_2 = 0 \wedge F_1^*] \vee F_2^*$$

where

$$\begin{array}{rcl} F_1^* & \equiv & a_1 \neq 0 \wedge (\exists x)[a_1 x + a_0 = 0 \wedge F] \\ F_2^* & \equiv & a_2 \neq 0 \wedge (\exists x)[a_2 x^2 + a_1 x + a_0 = 0 \wedge F] \end{array}$$

Thus the problem is reduced into two quantifier elimination problems: one for $F_1^*$ and the other for $F_2^*$ . One way for solving these problems might be to put the "parametric" roots

(expressions) into $F$ and eliminate them by rewriting. (as we have seen in the previous section on Linear Quantifier Elimination). But the following two theorems show us that we can bypass these two steps by utlizing the resultants and the slope resultants.

**Theorem 3 (Linear Case)** *Let $F'$ be the quantifier free formula in the variables $x_1, \ldots, x_r$ defined recursively by*

$$F' \equiv \begin{cases} R = 0 & \text{if } F \equiv B = 0 \\ R > 0 & \text{if } F \equiv B > 0, \deg_x(B) \text{ is even} \\ a_1 R > 0 & \text{if } F \equiv B > 0, \deg_x(B) \text{ is odd} \\ F_1' \vee F_2' & \text{if } F \equiv F_1 \vee F_2 \\ F_1' \wedge F_2' & \text{if } F \equiv F_1 \wedge F_2 \\ \neg F_1' & \text{if } F \equiv \neg F_1 \end{cases}$$

*where $R = \text{res}(a_1 x + a_0, B)$. Then we have*

$$F_1^* \Longleftrightarrow a_1 \neq 0 \wedge F' \quad \blacksquare$$

The following theorem shows a way to eliminate the existential quantifier from the formula $F_2^*$.

**Theorem 4 (Quadratic Case)** *Let $F^{(1)}$ be the quantifier free formula in the variables $x_1, \ldots, x_r$ defined recursively by*

$$F^{(1)} \equiv \begin{cases} R = 0 \wedge TS \leq 0 & \text{if } F \equiv B = 0 \\ R > 0 \wedge T > 0 \vee & \\ \quad R < 0 \wedge S > 0 \vee & \\ \quad T > 0 \wedge S > 0 & \text{if } F \equiv B > 0, \\ & \quad \deg_x(B) \text{ is even} \\ a_2 R > 0 \wedge a_2 T > 0 \vee & \\ \quad a_2 R < 0 \wedge a_2 S > 0 \vee & \\ \quad a_2 T > 0 \wedge a_2 S > 0 & \text{if } F \equiv B > 0 \\ & \quad \deg_x(B) \text{ is odd} \\ F_1^{(1)} \vee F_2^{(1)} & \text{if } F \equiv F_1 \vee F_2 \\ F_1^{(1)} \wedge F_2^{(1)} & \text{if } F \equiv F_1 \wedge F_2 \\ \neg F_1^{(1)} & \text{if } F \equiv \neg F_1 \end{cases}$$

*where*

$$\begin{aligned} R &= \text{res}(a_2 x^2 + a_1 x + a_0, B) \\ T &= \text{sres}_0(a_2 x^2 + a_1 x + a_0, B) \\ S &= \text{sres}_1(a_2 x^2 + a_1 x + a_0, B) \end{aligned}$$

*Let $F^{(2)}$ be the quantifier formula in the variables $x_1, \ldots x_r$ defined in the same way as $F^{(1)}$, except that $S$, $F_1^{(1)}$, and $F_2^{(1)}$ are replaced by $-S$, $F_1^{(2)}$, and $F_2^{(2)}$. Then we have*

$$F_2^* \Longleftrightarrow a_2 \neq 0 \wedge a_1^2 - 4a_2 a_0 \geq 0 \wedge [F^{(1)} \vee F^{(2)}] \quad \blacksquare$$

We summarize the above results in the following algorithm.

**Algorithm 3 (Quantifier Elimination)**

*Input* : A formula $F^*$ of the form

$$(\exists x \in \mathbb{R})[a_2x^2 + a_1x + a_0 = 0 \wedge F]$$

where $F$ is a quantifier free formula in $x_1, \ldots, x_r, x$ and
$a_2, a_1, a_0$ are polynomials over $x_1, \ldots, x_r$ such that
$a_2, a_1$ and $a_0$ do not have a common real zero in $\mathbb{R}^r$

*Output*: A quantifier-free formulat $\tilde{F}$ equivalent to $F$

(1) Apply on $F$ the recursive algorithm in Theorem 3, obtaining a quantifier free formula $F'$.
(2) Apply on $F$ the recursive algorithm in Theorem 4, obtaining two quantifier free formulas $F^{(1)}$ and $F^{(2)}$
(3) Obtain $\tilde{F}$ by putting together $F'$, $F^{(1)}$, and $F^{(2)}$ as follows:

$$\begin{aligned}\tilde{F} \quad \equiv \quad & a_2 = 0 \wedge a_1 \neq 0 \wedge F' \vee \\ & a_2 \neq 0 \wedge a_1^2 - 4a_2a_0 \geq 0 \wedge [F^{(1)} \vee F^{(2)}]\end{aligned}$$ ■

**Performance** If we allow the determinant symbol in the output, the computing time of the algorithm is *linear* in the length of the input. If not, the computing time is dominated by $N(n^{2r+1}\ell + n^{2r}\ell^2)$ where $N$ is the number of polynomials in the input formula, $r$ is the number of variables, $n$ is the maximum of the degrees for every variable, and $\ell$ is the maximum of the integer coefficient bit lengths. Experiments with implementation suggest that the algorithm is sufficiently efficient to be useful in practice.

### 14.3.3 Single Atomic Formula

Gonzales-Vega[Gonz98] gave algorithms for formulas with single atomic formula. The algorithms first compute a finite number of polynomials and select among all the possible sign conditions over these polynomials those making the considered formula true. The main mathematical tool used is the Sturm-Habicht sequence[Gonz89] which is essentially a careful adaptation of Sturm's sequence to the subresultant chain.

#### Restricted Problem Class

Let $P_n(\underline{a}, x)$ denote the polynomial $x^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0$. Then we are interested in the formulas of the following kind:

$$(Qx)P_n(\underline{a}, x) \ \rho \ 0$$

where $Q$ is either $\forall$ or $\exists$, and $\rho$ is a relational operator. One sees that the problem can be trivially reduced to the following two types:

$$\begin{aligned}\textbf{Type1}: \quad & \exists x \quad P_n(\underline{a}, x) = 0 \\ \textbf{Type2}: \quad & \exists x \quad P_n(\underline{a}, x) < 0\end{aligned}$$

for even $n$.

**Mathematical Tool: Sturm-Habicht**

Let $\mathbb{D}$ be an integral domain

**Definition 4**. *Let* $P = \sum_{k=0}^{p} a_k x^k$ *and* $Q = \sum_{k=0}^{q} b_k x^k$ *be polynomials in* $\mathbb{D}[x]$ *with* $\deg(P) \leq p$ *and* $\deg(Q) \leq q$. *The i-th formal principal subresultant coefficient.* $\mathbf{sres}_i(P, p, Q, q)$, *is the determinant of the following matrix:*

$$\overbrace{\qquad\qquad\qquad\qquad}^{p+q-2i}$$

$$\left(\begin{array}{ccccc} a_p & a_{p-1} & \cdots & & \\ & \ddots & & \ddots & \\ & & a_p & a_{p-1} & \cdots \\ b_1 & b_{q-1} & \cdots & & \\ & \ddots & & \cdots & \\ & & b_q & b_{q-1} & \cdots \end{array}\right) \begin{array}{l} \left.\vphantom{\begin{array}{c}a\\a\\a\\a\\a\end{array}}\right\} q-i \\ \left.\vphantom{\begin{array}{c}a\\a\\a\\a\\a\end{array}}\right\} p-i \end{array}$$

■

This is essentially the same as the usual definition of the principal subresultant coefficients (used in the projection of the Cylindrical Algebraic Decomposition method), except that it allows *formal* leading coefficients, that is, $\deg(P)$ and $\deg(Q)$ do not have to be exactly $p$ and $q$.

**Definition 5** *Let* $P, Q$ *be polynomials in* $\mathbb{D}[x]$. *Let* $v = \deg(P) + \deg(Q) - 1$. *The principal i-th Sturm-Habicht coefficient,* $\mathbf{stha}_i(P, Q)$, *is defined by*

$$\mathbf{stha}_i(P, Q) = (-1)^{(v-i)(v-i+1)/2} \mathbf{sres}_i(P, v+1, P'Q, v) \quad ■$$

The **stha** has a nice *specialization property*. The $\phi$ stand for the specialization of $a_i$ and $b_i$ to some real numbers such that the degrees stay the same. Then we obviously have the following:

$$\phi(\mathbf{stha}_i(P, Q)) = \mathbf{stha}_i(\phi(P), \phi(Q))$$

that is, the **stha** commutes with specialization.

**Definition 6** *Let* $a_0, a_1, \ldots, a_n$ *be a list of non zero elements in* $\mathbb{R}$. *We define:*

- $\mathbf{P}(a_0, a_1, \ldots, a_n)$ *as the number of sign permanence in the list* $a_0, a_1, \ldots, a_n$
- $\mathbf{V}(a_0, a_1, \ldots, a_n)$ *as the number of sign variations in the list* $a_0, a_1, \ldots, a_n$ ■

**Definition 7** *Let* $a_0, a_1, \ldots, a_n$ *be elements in* $\mathbb{R}$ *with* $a_0 \neq 0$. *Suppose that it is made of* $A_1, Z_1, A_2, Z_2, \ldots, A_t, Z_t$ *where each* $A_i$ *is a sequence of non-zeros and* $Z_i$ *is a sequence of zeros. Let* $k_i$ *be the length of* $Z_i$. *Let* $h_i$ *and* $t_i$ *be respectively, the sign of the head and the tail element of* $A_i$. *Then we define*

$$\mathbf{C}(a_0, a_1, \ldots, a_n) = \sum_{i=1}^{t} \mathbf{P}(A_i) - \sum_{i=1}^{t} \mathbf{V}(A_i) + \sum_{i=1}^{t-1} \epsilon_i$$

*where*

$$\epsilon_i = \begin{cases} 0 & \text{if } k_i \text{ is odd} \\ (-1)^{\frac{k_i}{2}} t_i h_{i+1} & \text{if } k_i \text{ is even} \end{cases} \quad ■$$

**Definition 8** *Let $P$ and $Q$ be polynomials in $\mathbb{R}[x]$. Then let*

$$c_\epsilon(P;Q) = \text{card}(\{\alpha \in \mathbb{R} | P(\alpha) = 0, \quad \text{sign}(Q(\alpha)) = \epsilon\})$$

*where $\epsilon$ is a sign $(+,-,0)$.* ■

**Theorem 5** *Let $P$ and $Q$ be polynomials in $\mathbb{R}[x]$ with $\deg(P) = p$. Then we have*

$$\mathbf{C}(\mathbf{stha}_p(P,Q), \ldots, \mathbf{stha}_0(P,Q)) = c_+(P;Q) - c_-(P;Q) \quad \blacksquare$$

### Method

Let us tackle now the two types of quantifier elimination problems mentioned above. From now, let

$$\mathbf{C}(P,Q) = \mathbf{C}(\mathbf{stha}_p(P,Q), \ldots, \mathbf{stha}_0(P,Q))$$

**Type 1** From the good specialization property of **stha** and Theorem 5, we immediately obtain

$$\exists x P_n(\underline{a}, x) = 0 \Longleftrightarrow \mathbf{C}(P_n, 1) > 0$$

Now we only need to "expand" the right hand side by checking all the $3^{n+1}$ possible sign conditions over the polynomials $\mathbf{stha}_i(P_n, 1)$ and keeping those making $C > 0$.

**Type 2** Let $I_n^i$ denote the formula stating that the polynomial $P_n(a,x)$ has a real root with multiplicity $i$. Obviously we have

$$\exists x P_n(\underline{a}, x) < 0 \Longleftrightarrow \bigvee_{i=1,\text{odd}}^{n-1} I_n^i(\underline{a})$$

So we only need to find a quantifier-free formula for $I_n^i$. From the definition of multiplicity, we immediately have

$$\begin{aligned} I_n^i(\underline{a}) &\iff (\exists x)[P_n^{(0)}(\underline{a},x) = \ldots = P_n^{(i-1)}(\underline{a},x) = 0 \\ &\quad \wedge P_n^{(i)}(\underline{a},x) \neq 0] \\ &\iff (\exists x)[\sum_{k=0}^{i-1}(P_n^{(k)}(\underline{a},x))^2 = 0 \wedge P_n^{(i)}(\underline{a},x) \neq 0] \\ &\iff (\exists x)[\sum_{k=0}^{i-1}(P_n^{(k)}(\underline{a},x))^2 = 0 \wedge (P_n^{(i)}(\underline{a},x))^2 > 0] \end{aligned}$$

Since square of real number is non-negative, we immediately see from Theorem 5 that

$$I_n^i(\underline{a}) \Longleftrightarrow \mathbf{C}(\sum_{k=0}^{i-1}(P_n^{(k)}(\underline{a},x))^2, (P_n^{(i)})^2) > 0$$

Thus, finally we have

$$\exists x P_n(\underline{a}, x) < 0 \Longleftrightarrow \bigvee_{i=1,\text{odd}}^{n-1} \mathbf{C}(\sum_{k=0}^{i-1}(P_n^{(k)}(\underline{a},x))^2, (P_n^{(i)})^2) > 0$$

Again we will have to expand the right hand side by checking all the possible sign conditions over the **stha** polynomials and keeping those making $\mathbf{C} > 0$.

## 14.4 Approximate Methods

By now, it is well understood both theoretically and experimentally that the full exact quantifier elimination is intrinsically difficult, posing a formidable barrier to efforts for devising general but efficient methods. One response to this is specialization, as we have surveyed in the previous section. In this section, we see another response: *approximation*.

In many applications (in particular in science and engineering), an approximate answer is acceptable. In fact, sometimes, it is even meaningless to think of exact answers because the input itself often contain non-exactness.

This is a good news for algorithm designers, since experience shows that the enormous complexity of quantifier elimination mainly stems from the insistence on exactness. If we are willing to compromise this, then we might be able to devise methods that can handle large problems arising from real-life applications.

This does not mean that approximation will give a better asymptotic worst case bound. It might be the case that the approximated methods have the same bound, or even worse. But there is evidence that the approximated methods are much more efficient for a large class of moderate size inputs.

The word "approximate" carries different meanings in different contexts. In this section, we survey two different interpretations and methods for them: *generic* and *volume-approximate*.

### 14.4.1 Generic Quantifier Elimination

One often observes that exact algorithms spend most of their computing time in analyzing and taking care of non-generic cases. But the end-users of the methods are often interested only in generic (regular) cases. In such situations, the exact methods wasted computing resource to produce unnecessary information. It will be better to have a method that works only on generic cases.

This observation lead Hong to interpret approximatedness as being *correct except for some non- generic cases*. The non-generic cases usually form a measure-zero set, and thus generically correct answers are different from the exact answers only by measure zero sets. Thus, it can be also viewed as being *correct up to measure zero*.

#### Problem

We begin by giving a precise statement of the generic quantifier elimination problem. Let $F(x_1, \ldots, x_f)$ be a formula. Let $S(F)$ stand for the solution set of $F$, that is, the set $\{p \in \mathbb{R}^f \mid F(p) \text{ is true}\}$. We will define two notions: *generic quantifiers* and *generic equivalence*.

**Definition 9 (Generic Quantifiers)** *Let $F$ be a formula with one free variable $x$.*

- *The* generic universal quantifier, $\dot{\forall}$, *is defined by: $\dot{\forall}xF(x)$ is true iff $S(\neg F)$ is of measure zero*
- *The* generic existential quantifier, $\dot{\exists}$, *is defined by: $\dot{\exists}xF(x)$ is true iff $S(F)$ is of measure zero* ■

Intuitively, the generic universal quantifier can be read as "for almost all", and the generic

existential quantifier can be be read as "for sufficiently many". As expected, the following properties hold for the generic quantifiers.

**Proposition 2 (Commutativity)**

- $\dot{\forall}x\dot{\forall}yF(x,y) \Longleftrightarrow \dot{\forall}y\dot{\forall}xF(x,y)$
- $\dot{\exists}x\dot{\exists}yF(x,y) \Longleftrightarrow \dot{\exists}y\dot{\exists}xF(x,y)$ ■

**Proposition 3 (Negation)**

- $\neg\dot{\forall}xF(x) \Longleftrightarrow \dot{\exists}x\neg F(x)$
- $\neg\dot{\exists}xF(x) \Longleftrightarrow \dot{\forall}x\neg F(x)$ ■

The "strength" of the four quantifiers (two exact and two generic) form the following spectrum.

**Proposition 4 (Spectrum of quantification strength)**

$$\forall xF(x) \Longrightarrow \dot{\forall}xF(x) \Longrightarrow \dot{\exists}xF(x) \Longrightarrow \exists xF(x) \quad ■$$

**Definition 10 (Generic equivalence)** *Two formulas $F$ and $G$ with free variables $x_1, \ldots, x_f$ are generically equivalent iff the sentence*

$$\dot{\forall}x_1, \ldots \dot{\forall}x_f[\ F(x_1, \ldots, x_f) \Longleftrightarrow G(x_1, \ldots, x_f)\ ]$$

*is true* ■

Intuitively, two formulas are generically equivalent if their solution sets are "almost the same". Finally we are ready to state the generic quantifier elimination problem.

**Problem**: Device an algorithm with the specification:

*In:* A formula $F(x_1, \ldots, x_f)$ involving only generic quantifiers
*Out:* A formula $G(x_1, \ldots, x_f)$ such that it involves no (generic) quantifiers and $F$ and $G$ are generically equaivalent. ■

**Method**

A natural way is to take an existing exact algorithm and turn it into a generic one. We will do so with the method of (partial) cylindrical algebraic decomposition [Coll75][Hong90a][Coll91].

**Idea 1: No algebraic numbers**. Since the generic quantifier elimination ignores measure zero sets, we do not need to construct stacks over even-indexed cells (because they are of measure zero). This results in a significant reduction of computing time since we can completely eliminate the heavy real algebraic number computations (which are needed for working with even indexed cells).

**Idea 2: Smaller Projection**. Further, the projection polynomials are non-zero throughout odd indexed cells. Thus we can reduce the size of projection by removing the polynomials

that are put in there to ensure delineability over even indexed cells. Precisely, we have the following:

**Definition 11 (Generic Projection)** *Let $A$ be a finite subset of $\mathbb{Z}[x_1, \ldots, x_r]$. The generic projection of $A$, written as $\mathrm{GProj}(A)$, is defined as:*

$$\begin{aligned} \mathrm{GProj}(A) &= L \cup R \cup D \\ L &= \{lc(a) \mid a \in A\} \\ R &= \{\mathrm{psc}_0(a_1, a_2) \mid a_1, a_2 \in A\} \\ D &= \{\mathrm{psc}_0(a, a') \mid a \in A\} \quad \blacksquare \end{aligned}$$

**Theorem 6 (Generic Projectiong gives Delineability)** *Let $A$ be a finite subset of $\mathbb{Z}[x_1, \ldots, x_r]$, $r \geq 2$. Let $c$ be a non-empty connected subset of $\mathbb{R}^{r-1}$ such that for all $J \in \mathrm{GProj}(A)$ and for all $p \in c$ we have $J(p) \neq 0$. Then the zero of $A$ delineates the cylinder over $c$.* $\blacksquare$

**Idea 3: No square-free factorization needed**. Stack construction consists of the following steps:

(1) Evaluate the projection polynomials on a sample sample, obtaining a set of univariate polynomials.
(2) Factorize the univariate polynomials, obtaining square-free and relatively prime factors.
(3) Isolate the real roots of the univariate polynomials.

The step (2) is often most time-consuming. But in generic quantifier elimination, we can drop this step since:

- During the projection phase, the projection polynomials are already made to be square-free and relatively prime. (If not, their projection polynomials will identically vanish and provide no information).
- Since the leading coefficients of the polynomials do not vanish on the sample point, the evaluated univariate polynomials will also be square-free and relatively prime.

**Idea 4: Sturm sequence for free**. Again utilizing the fact that the leading coefficients of the polynomials do not vanish on the sample point, we can obtain a Sturm sequences for them almost for free if we have computed the projection polynomials by subresultant sequence or Sturm-Habicht sequence[Gonz89]. This is because the sequences have a good specialization property when the leading coefficients do not vanish. Thus the obtained Sturm sequence can be used for real root isolation.

Let us make this idea more precise for a subresultant sequence. We first need some notation and notions.

- Let $\mathbb{D}$ denote an integral domain. When necessary, it will also be assumed that $\mathbb{D}$ is ordered.
- Let $A_1, A_2 \in \mathbb{D}[x]$ be such that $\deg(A_1) = n_1$, $\deg(A_2) = n_2$, and $n_1 \geq n_2 \geq 0$.
- A *polynomial remainder sequence* (prs) of $A_1$ and $A_2$ is a sequence $A_1, \ldots, A_r$ satisfying

$$e_i A_i = Q_i A_{i+1} + f_i A_{i+2}$$

where $A_i, Q_i \in \mathbb{D}[x]$, $e_i, f_i \in \mathbb{D}$, $\deg(A_{i+1}) > \deg(A_{i+2})$, $A_{r+1} = 0$ and $e_i f_i \neq 0$.

- A *negative* prs of $A_1$ and $A_2$ is a prs such that $e_i f_i < 0$ for every $i$
- A *Sturm sequence* of $A \in \mathbb{D}[x]$ is a negative prs of $A$ and $A'$.
- The $k$-th *Sylvester matrix* of $A_1$ and $A_2$, $0 \leq k \leq n_2$, is the matrix whose rows are the coefficients of the polynomials:

$$x^{n_2-1-k}A_1, \ldots, A_1, x^{n_1-1-k}A_2, \ldots, A_2$$

  written in the basis: $x^{n_1+n_2-1-k}, \ldots, 1$
- The $k$-th *subresultant* of $A_1$ and $A_2$, $0 \leq k \leq n_2$, written as $S_k$, is defined by

$$S_k = \sum_{i=0}^{k} \det(M_k^{(i)})x^i$$

  wher $M_k^{(i)}$ is the submatrix of the $k$-th Sylviester matrix of $A_1$ and $A_2$ obtained by taking the first $n_1 + n_2 - 1 - 2k$ columns and the $(n_1 + n_2 - k - i)$-th column.
- The sequence $A_1, A_2, S_{n_2-1}, S_{n_3-1}, \ldots, S_{n_{r-1}-1}$ is called the *first kind subresultant* prs of $A_1$ and $A_2$ ■

Let $\phi$ stand for an evaluation homomorphism that preserves the degree (that is, the leading coefficients do not vanish under it). Let $A_1, A_2 \in \mathbb{D}[y]$. The following theorem shows how to obtain the first kind subresultant prs of $\phi(A_1)$ and $\phi(A_2)$ from that of $A_1$ and $A_2$.

**Theorem 7 (Homomorphism on first kind)** *Let $A_1, A_2, \ldots, A_r$ be the first kind subresultant* prs *of $A_1$ and $A_2$. Let $\tilde{A}_1, \ldots, \tilde{A}_{\tilde{r}}$ be the sequence obtained from $\phi(A_1), \ldots, \phi(A_r)$ by deleting the vanishing ones, and deleting the second one in case there are two of them with the same degree. (At most two can be of the same degree.) More specifically if $\deg(\phi(A_k)) = \deg(\phi(A_j))$ for $i < j$, then we delete $\phi(A_j)$. Then the sequence is the first kind subresultant* prs *of $\phi(A_1)$ and $\phi(A_2)$* ■

Let $\tilde{A}_1, \tilde{A}_2, \ldots, \tilde{A}_r$ be the first kind subresultant prs of $\tilde{A}$ and $\tilde{A}'$, obtained as described above. Let $\tilde{n}_i = \deg(\tilde{A}_i)$, $\delta_i = \tilde{n}_i - \tilde{n}_{i+1}$, and $\tilde{c}_i$ be the leading coefficient of $\tilde{A}_i$. Then the following theorem shows how to obtain a Sturm sequence of $\tilde{A}$ from the first kind resultant prs of $\tilde{A}$ and $\tilde{A}'$.

**Theorem 8 (Sturm sequence from sres** *prs***)** *Let*

$$\begin{array}{rcl}
\lambda_i & = & \text{the sign of } \tilde{c}_i \text{ for } 2 \leq i \leq r-1 \\
\mu_i & = & \lambda_{i+1}^{\delta_i+1} \text{ for } 1 \leq i \leq r-2 \\
\rho_1 & = & 1 \\
\rho_i & = & \lambda_i^{\delta_{i-1}} \rho_{i-1}^{\delta_{i-1}-1} \text{ for } 2 \leq i \leq r-2 \\
\nu_1 & = & 1 \\
\nu_i & = & -\lambda_i(-\rho_i)^{\delta_i} \text{ for } 2 \leq i \leq r-2 \\
\sigma_1 & = & 1 \\
\sigma_2 & = & 1 \\
\sigma_i & = & -\mu_{i-2}\nu_{i-2}\sigma_{i-2} \text{ for } 3 \leq i \leq r \\
\tilde{A}_i & = & \sigma_i \tilde{A}_i \text{ for } 1 \leq i \leq r
\end{array}$$

*Then $\tilde{A}_1, \ldots, \tilde{A}_r$ is a Sturm sequence of $\tilde{A}$* ■

Now we present an algorithm for generic quantifier elimination, based on the above ideas. The general structure of the algorithm is based on that of the partial cylindrical algebraic decomposition [Hong90a][Coll91]. We use some standard terminology from there.

**Algorithm 4 (Generic Quantifier Elimination)**

*Input*: A formula $F' = (Q_{f+1}x_{f+1}) \cdots (Q_r x_r) F(x_1, \ldots, x_r)$
where $Q_i$ is either $\forall$ or $\exists$ and $F$ is a quantifier free formula.
*Output*: A quantifier-free formula $G$ which is generically equivalent to $F$.

(1) [Projection]
Factorize the polynomials occurring in the formula $M$.
For $k = 1, \ldots, r$, let $P_k$ be the set of the $i$-level factors.
For $k = r, \ldots, 2$ do
  (a) $J \longleftarrow \mathrm{GProj}(P_k)$. *Use the first kind subresultant polynomial remainder sequence to compute* psc's
  (b) Factorize the polynomials in $J$
  (c) For $i = 1, \ldots, k-1$, append the $i$-level factors to $P_i$ if they are not already there.

(2) [Stack Construction]
Initialize the tree as a single node whose truth is ? and whose sample point is ().
While there is a candidate node do
  (a) Choose a candidate node $c$. Let $k$ be the level of the node and let $s = (s_1, \ldots, s_k)$ be its sample point
  (b) Evaluate the polynomials in $P_{k+1}$ on $s$ obtaining univariate polynomials
  (c) Obtain Sturm sequences of the polynomials of the univariate polynomials by specializing the first kind subresultant polynomials remainder sequences computed during projection.
  (d) Using the Sturm sequences, find rational numbers $t_1, \ldots, t_\ell$ such that $t_1 < \alpha_1 < t_2 < \alpha_2 < \cdots < \alpha_{\ell-1} < t_\ell$ where $\alpha_1, \ldots, \alpha_{\ell-1}$ are the real roots of the univariate polynomials
  (e) Attach $\ell$ nodes, say, $c_1, \ldots, c_\ell$, to the node $c$ where the sample point of $c_i$ is set to be $(s_1, \ldots, s_k, t_i)$
  (f) For each child $c_i$, determine the signs of the polynomials in $P_{k+1}$ on it
  (g) For each child $c_i$, try to determine its truth by using the signs
  (h) Determine the truth values of $c$ if possible, and if so, the truth values of as many ancestors of $c$ as possible, removing from the tree the subtree of each cell whose truth value is thus determined

(3) [Solution Formula Construction]
By using the method described in [Hong98], construct a quantifier free formula $G$ such that it is true on all the true cells and false on all the false cells. ■

### 14.4.2 Volume Approximate Quantifier Elimination

Hong introduced the notion of volume approximate quantifier elimination and a method for it. Let us first clarify what is meant by "volume approximate" quantifier elimination. It was further improved by Andreas Neubacher and completed by Stefan Ratschan[Rats02].

### Problem

We define the volume approximate quantifier elimination problem as follows. Devise an algorithm for

**In:** $F$, a formula
$\epsilon$, a positive real number
**Out:** $N$, a quantifier-free formula necessary to $F$
$S$, a quantifier-free formula sufficient to $S$ such that $V(N) - V(S) \leq \epsilon$

where $V(G)$ stands for the volume of the set defined by the formula $G$. Usually, $N$ and $S$ are required to belong to a small subset of the full language. For instance, they are required to be a disjunctive normal form of linear inequalities, that is, a collection of convex polytopes. Or one can put even stronger restriction on them to be a disjunctive normal form of "de-coupled" linear inequalities, that is, a collection of boxes aligned with the coordinate axis. Throughout this section, we will restrict our discussion to the case when $N$ and $S$ are required to be a collection of closed boxes. Further, we will require that the boxes do not overlap. We will call such a collection a *box-set*.

Note that a box can viewed as a set of points or a formula that defines the set. We will use these two views interchangeably. Thus, sometimes we will speak of "union" of boxes and some other times we will speak of "disjunction" of boxes. Obviously these are the same operations.

The boxes that we will use are closed ones. This causes some complications in defining the notion of "disjointness". We will not go through the technical details in this tutorial. Intuitively, we will interpret that two boxes are disjoint if their intersection is either empty or belongs to the boundaries.

### Method

One natural idea is to approximate each atomic formula by two box-sets: one as a necessary condition, the other as a sufficient condition. Then carry out the logical operations (disjunction, quantification, etc) on the box-sets. These logical operations are much easier, though not trivial, to do on box-sets than on arbitrary formulas. Also they can be done *exactly*.

A question naturally arises: How accurately should we approximate each atomic formula? One can take an "eager" approach. That is to carry out an *error-analysis* to estimate a lower bound on the accuracy that will ensure that, after the logical operations on the box-sets, the volume difference is smaller than $\epsilon$. The advantage of this scheme is that once the error-analysis is done, we need to carry out the approximation of atomic formulas and the logical operations on them only once. But there are two serious disadvantages/difficulties.

- The error-analysis expects and prepares for the "worst-case" situation. Thus, in average, it can induce much more work than actually needed.
- A reasonable error-analysis is very difficult to do for quantifications. A natural (geometric) method gives a totally useless bound (too big). One can give better analysis using a root-separation theorem, gap theorem, etc. But such analysis is computationally too costly.

A better way is to take the so-called "lazy" approach. This means to compute only when necessity arises. In that sense it is "demand-driven". Thus, at first, we carry out a very "coarse" box-approximation of atomic formulas. If, after logical operations on the boxes,

the volume difference is already less than $\epsilon$, then we can stop. If not, we can refine the initial box-approximation of atomic formulas and iterate the same process until the volume difference is smaller than $\epsilon$.

Further, during the execution of the method, we would like to utilize any intermediate information as soon as they become available, in particular for pruning/reducing the "search-space". As an example, consider the formula $F \equiv F_1 \wedge F_2$. Now we would like to approximate it by a necessary box-set $N$ and a sufficient box-set $S$. One elegant way is to approximate $F_1$ and $F_2$ independently and and carry out the conjunction on the resulting box-sets $N_1, S_1$ and $N_2, S_2$. But here, the result of the approximation of $F_1$ is not used during the approximation of $F_2$. This can cause useless work. For instance, it might the case that $N_1$ is empty (that is false). Then obviously the result $N$ is also false, no matter what $N_2$ is. Thus a better way is to use $N_1$ and $S_1$ during the approximation of $F_2$.

Now we turn the ideas discussed above into algorithms. We will use the following notations:

- $+$ *disjoint disjunction* (or *sum*). $B_1 + B_2$ is the same as $B_1 \vee B_2$ except that it also states that $B_1$ and $B_2$ are disjoint. Theus, this operation can be trivially done by collecting all the boxes in the two box-sets
- $\times$ *independent conjunction* (or *product*). $B_1 \times B_2$ is the same as $B_1 \wedge B_2$ except that it also states that $B_1$ and $B_2$ are independent (they do not share variables). Thus, it is also trivial to carry out.

A technical remark. Though in the end, we would like to obtain a necessary box-set and a sufficient box, it is more convenient/efficient to keep track of "yes box-set", "unknown box set", "no box-set" where the "yes box-set" is inside the solution set of $F$, the "no box-set" is outside the solution set of $F$, the "unknown box-set" is not known yet whether it is inside or outside. These will be denoted as $B_y$, $B_u$, $B_n$ respectively.

**Algorithm 5**

---

$$(N, S) \longleftarrow VolumeApproxQE(F)$$

*Input*: $F$ is a formula
*Output*: $N$ is a necessary condition of $F$ and $S$ is a sufficient condition of $F$ such that $V(N) - V(S) \leq \epsilon$.

(1) $B_y \longleftarrow$ the empty set (which represents the logical false)
$B_n \longleftarrow$ the empty set
$B_u \longleftarrow$ the whole free variable space

(2) While $V(B_u) > \epsilon$ repeat
$(B_{uy}, B_{uu}, B_{un}) \longleftarrow ApproxFormula(F, B_u)$
$B_y \longleftarrow B_y + B_{uy}$
$B_u \longleftarrow B_n + B_{un}$
$B_u \longleftarrow B_{uu}$

(3) $N \longleftarrow B_y + B_u$
$S \longleftarrow B_y$ ■

---

Now we describe the sub-algorithm *ApproxFormula*. We will use the following notational convention: $B_{pq}$ where $p$, $q$ can be one of $y$, $n$, $u$, $*$, which respectively stand for yes, no, unknown, any of them. The first index $p$ tells us the status of the box with respect to a formula $F_1$ and the second index $q$ to $F_2$ . For instance $B_{yu}$ means that the box-set is a yes

box-set with respect to $F_1$ and a unknown box-set with respect to $F_2$.

**Algorithm 6**

---

$$(B_y, B_u, B_n) \longleftarrow ApproxFormula(F, B)$$

*Input*: $F$ is a formula and $B$ is a box-set
*Output*: $B_y$, $B_u$, $B_n$ are box-sets that partition $B$ such that $B_y \Longrightarrow F$ and $B_n \Longrightarrow \neg F$

(1) $F$ is an atomic formula
$(B_y, B_u, B_n) \longleftarrow ApproxAtomic(F, B)$

(2) $F \equiv F_1 \wedge F_2$
$(B_{y*}, B_{u*}, B_{n*}) \longleftarrow ApproxFormula(F_1, B)$
$(B_{yy}, B_{yu}, B_{yn}) \longleftarrow ApproxFormula(F_2, B_{y*})$
$(B_{uy}, B_{uu}, B_{un}) \longleftarrow ApproxFormula(F_2, B_{u*})$
$B_y \longleftarrow B_{yy}$
$B_u \longleftarrow B_{yu} + B_{uy} + B_{uu}$
$B_n \longleftarrow B_{n*} + B_{yn} + B_{un}$

(3) $F \equiv F_1 \vee F_2$
$(B_{y*}, B_{u*}, B_{n*}) \longleftarrow ApproxFormula(F_1, B)$
$(B_{ny}, B_{nu}, B_{nn}) \longleftarrow ApproxFormula(F_2, B_{n*})$
$(B_{uy}, B_{uu}, B_{un}) \longleftarrow ApproxFormula(F_2, B_{u*})$
$B_y \longleftarrow B_{y*} + B_{ny} + B_{uy}$
$B_u \longleftarrow B_{nu} + B_{un} + B_{uu}$
$B_n \longleftarrow B_{nn}$

(4) $F \equiv \neg F$
$(B_{y*}, B_{u*}, B_{n*}) \longleftarrow ApproxFormula(F_1, B)$
$B_y \longleftarrow B_{n*}$
$B_u \longleftarrow B_{u*}$
$B_n \longleftarrow B_{y*}$

(5) $F \equiv F_1 \Longrightarrow F_2$
$(B_{y*}, B_{u*}, B_{n*}) \longleftarrow ApproxFormula(F_1, B)$
$(B_{yy}, B_{yu}, B_{yn}) \longleftarrow ApproxFormula(F_2, B_{y*})$
$(B_{uy}, B_{uu}, B_{un}) \longleftarrow ApproxFormula(F_2, B_{u*})$
$B_y \longleftarrow B_{n*} + B_{yy} + B_{uy}$
$B_u \longleftarrow B_{yu} + B_{un} + B_{uu}$
$B_n \longleftarrow B_{yn}$

(6) $F \equiv F_1 \Longleftrightarrow F_2$
$(B_{y*}, B_{u*}, B_{n*}) \longleftarrow ApproxFormula(F_1, B)$
$(B_{yy}, B_{yu}, B_{yn}) \longleftarrow ApproxFormula(F_2, B_{y*})$
$(B_{ny}, B_{nu}, B_{nn}) \longleftarrow ApproxFormula(F_2, B_{n*})$
$B_y \longleftarrow B_{yy} + B_{nn}$
$B_u \longleftarrow B_{u*} + B_{yu} + B_{nu}$
$B_n \longleftarrow B_{yn} + B_{ny}$

(7) $F \equiv \forall x F_1$ where $x$ is a vector of $n$ variables
$(B_{y*}, B_{u*}, B_{n*}) \longleftarrow ApproxFormula(F_1, B \times \mathbb{R}^n)$
$B_y \longleftarrow \forall x B_{y*}$

$B_n \longleftarrow \exists x B_{n*}$
$B_u \longleftarrow B - (B_y + B_n)$

(8) $F \equiv \exists x F_1$ where $x$ is a vector of $n$ variables
$(B_{y*}, B_{u*}, B_{n*}) \longleftarrow ApproxFormula(F_1, B \times \mathbb{R}^n)$
$B_y \longleftarrow \exists x B_{y*}$
$B_n \longleftarrow \forall x B_{n*}$
$B_u \longleftarrow B - (B_y + B_n)$

---

In the cases (7) and (8), the notation "$B \times \mathbb{R}^n$" means to embed each box in $B$ into the $m+n$ dimensional space by letting the $n$ new variables to range over $(-\infty, \infty)$.

Correctness of this algorithm is not difficult to prove and is left to the reader. Now we give brief and informal descriptions of the sub-algorithms:

| | |
|---|---|
| •*ApproxAtomic* | approximation of atomic formula |
| •$-$ | subtraction of box-sets |
| •$\vee$ | disjunciton of box-sets |
| •$\wedge$ | conjuncton of box-sets |
| •$\exists$ | existential quantification of box-sets |
| •$\forall$ | universal quantification of box-sets |

In order to keep the presentation of the ideas simple, we will intentionally avoid optimizations, except straightforward ones. But currently various optimizations, improvements and different methods are being researched.

**Approximating Atomic Formula**: Consider an atomic formula $P > 0$ and a box-set $B$. Without losing generality let us assume that $B$ consists of just one box. If not, we can apply the method described below to each box in the box-set (or just one box from it). Our goal is to partition $B$ into three box-sets $B_y$, $B_u$, $B_n$ such that every point of $B_y$ satisfies $P > 0$ and no point of $B_n$ satisfies $P > 0$.

We first check if the whole $B$ satisfies the atomic formula. This can be done by computing an interval $I$ that bounds the range of the polynomial $P$ on $B$, that is, $P(B) \subseteq I$. The range estimation can be done by using various methods from interval mathematics. If $I > 0$, then we can set $B_y$ to be $B$ and the others empty. If $I \leq 0$, then we can set $B_n$ to be $B$ and the other empty. Otherwise, we split the box $B$ into several boxes and do the same on each box.

But this method can be very inefficient because in general it will result in numerous small boxes (exponential in the number of variables). A better approach is to adapt the *tightening* method of Hong[Hong94]. The method roughly works as follows. Let $P \in \mathbb{R}[x_1, \ldots, x_r]$ and $B$ is a box in the $r$-dimensional space. We choose a variable, say $x_k$. We view $P$ as a univariate polynomial in $x_k$ . Then the coefficients are polynomials in the other variables. We, using interval methods, compute intervals that bound the coefficients on the box

$$B_1 \times \ldots \times B_{k-1} \times B_{k+1} \times \ldots \times B_r$$

As a result, we obtain a univariate polynomial in $x_k$ with interval coefficients. Next, we compute the "root intervals" of the polynomial, say $R_1, \ldots, R_\ell$. Let $C_1, \ldots, C_u$ be the intervals that form the complement of the root intervals with respect to $B_k$. Let $B^i$ be the box:

$$B_1 \times \ldots \times B_{k-1} \times C_i \times B_{k+1} \times \ldots \times B_r$$

Then the sign of $P$ is constant throughout each $B^i$, and the sign is either positive or negative. If $P$ is positive on $B^i$, then we put $B^i$ into the box-set $B_y$, and if negative, into the box-set

$B_n$. We put the remaining parts of the box $B$ into $B_u$. Specifically we put into $B_u$ the boxes

$$B_1 \times \ldots \times B_{k-1} \times R_i \times B_{k+1} \times \ldots \times B_r$$

**Subtraction of Box-sets**: Let $B$ and $C$ be two box-sets. We would like to compute a box-set $D$ such that $D \Longleftrightarrow B - C$, that is, $B \wedge \neg C$.

If $B$ is empty then $D$ is empty. If $C$ is empty then $D = B$. Thus assume that $B$ is not empty and $C$ is not empty. Let $B = B^1 + B^*$ and $C = C^1 + C^*$. Observe:

$$\begin{array}{rcl} B - C & \Longleftrightarrow & (B^1 + B^*) - C \\ & \Longleftrightarrow & (B^1 - C) + (B^* - C) \\ & \Longleftrightarrow & (B^1 - (C^1 + C^*)) + (B^* - C) \\ & \Longleftrightarrow & ((B^1 - C^1) - C^*) + (B^* - C) \end{array}$$

Thus, the subtraction between two box-sets is reduced to the subtraction between two boxes: $B^1 - C^1$.

So let us now find out how to do subtraction between two boxes. Let $B$ consist of only one box and $C$ also consist of only one box. Let $B = B_1 \times B_*$ and let $C = C_1 \times C_*$. Observe:

$$\begin{array}{rcl} B - C & \Longleftrightarrow & (B_1 \times B_*) - (C_1 \times C_*) \\ & \Longleftrightarrow & (B_1 \times B_*) \wedge \neg(C_1 \times C_*) \\ & \Longleftrightarrow & (B_1 \times B_*) \wedge (\neg C_1 \vee \neg C_*) \\ & \Longleftrightarrow & (B_1 \times B_*) \wedge (\neg C_1 + (C_1 \wedge \neg C_*)) \\ & \Longleftrightarrow & ((B_1 \wedge \neg C_1) \times B_*) + ((B_1 \wedge C_1) \times (B_* \wedge \neg C_*)) \\ & \Longleftrightarrow & ((B_1 - C_1) \times B_*) + ((B_1 \wedge C_1) \times (B_* - C_*)) \end{array}$$

Thus the subtraction between two boxes is reduced to the subtraction of two intervals $(B_1 - C_1)$ and the conjunction of two intervals $(B_1 \wedge C_1)$. These are trivial to do.

**Disjunction of Box-sets**: Let $B$ and $C$ be two box-sets. We would like to compute a box-set such that $D \Longleftrightarrow B \vee C$. We can immediately reduce this to the subtraction problem as:

$$B \vee C \Longleftrightarrow B + (C - B)$$

**Conjunction of Box-sets** Let $B$ and $C$ be two box-sets. We would like to compute a box-set $D$ such that $D \Longleftrightarrow B \wedge C$. Let $B = B^1 + \cdots + B^\mu$ and let $C = C^1 + \cdots + C^\nu$. We have

$$\begin{array}{rcl} B \wedge C & \Longleftrightarrow & (+_{i=1}^{\mu} B^i) \wedge (+_{j=1}^{\nu} C^j) \\ & \Longleftrightarrow & +_{i=1}^{\mu} +_{j=1}^{\nu} (B^i \wedge C^j) \\ & \Longleftrightarrow & +_{i=1}^{\mu} +_{j=1}^{\nu} (\times_{k=1}^{n} B_k^i \wedge \times_{k=1}^{n} C_k^j) \\ & \Longleftrightarrow & +_{i=1}^{\mu} +_{j=1}^{\nu} \times_{k=1}^{n} (B_k^i \wedge C_k^j) \end{array}$$

Thus the conjunction of box-sets is reduced to the conjuntions of intervals, which are trivial to do.

**Existential Quantifier Elimination on Box-sets**: Consider $F \equiv \exists x B$ where $x$ is a vector of $n$ variables and $B$ is a box-set in the $\mathbb{R}^{m+n}$ dimensional space. First note that $B$ is a

disjunction of boxes, say $B^1, \ldots, B^\ell$. Since the disjunction commutes with the existential quantification, we have

$$F \Longleftrightarrow \bigvee_{i=1}^{\ell} \exists x B^i$$

Now let $B^i = B^i_1 \wedge \cdots \wedge B^i_{m+n}$ where $B^i_j$ is an interval in the $x_j$-line. Let $\underline{B^i}$ intervals corresponding to the free variables and $\overline{B^i}$ be the conjunction of the $n$ intervals corresponding to the bound variables. Note that

$$\exists x B^i \Longleftrightarrow \exists x \underline{B^i} \wedge \overline{B^i} \Longleftrightarrow \underline{B^i} \wedge \exists x \overline{B^i} \Longleftrightarrow \underline{B^i}$$

So we trivially have

$$F \Longleftrightarrow \bigvee_{i=1}^{\ell} \underline{B^i}$$

Thus the problem is reduced to the disjunction problem.

**Universal Quantifier Elimination on Box-sets**: Consider $F \equiv \forall x B$. One way is to turn this into existential problem through double negation. But there is a better way. Let $B^1$ be a box in the box-set $B$ and let $B$ be the box-set consisting of the remaining boxes. Now observe:

$$\begin{array}{rcl}
F & \equiv & \forall x B \\
& \Longleftrightarrow & \forall x [B^1 \vee B^*] \\
& \Longleftrightarrow & \forall x [(\underline{B^1} \wedge \overline{B^1}) \vee B^*] \\
& \Longleftrightarrow & \forall x [(\underline{B^1} \vee B^*) \wedge (\overline{B^1} \vee B^*)] \\
& \Longleftrightarrow & \forall x [\underline{B^1} \vee B^*] \wedge \forall x [\overline{B^1} \vee B^*] \\
& \Longleftrightarrow & (\underline{B^1} \vee \forall x B^*) \wedge \forall x [\overline{B^1} \vee B^*] \\
& \Longleftrightarrow & (\underline{B^1} \wedge \forall x [\overline{B^1} \vee B^*]) \vee (\forall x B^* \wedge \forall x [\overline{B^1} \vee B^*]) \\
& \Longleftrightarrow & (\underline{B^1} \wedge \forall x [\overline{B^1} \vee B^*]) \vee \forall x B^*
\end{array}$$

Note that we have divided the formula $F$ into two smaller formulas. This suggests a recursive-divide-conquer algorithm. For this goal, let us rewrite the above observation in a slightly more general way (so that we can do recursion):

$$\begin{array}{rl}
& L \wedge \forall x [U \vee B] \\
\Longleftrightarrow & (L \wedge \underline{B^1} \wedge \forall x [U \vee \overline{B^1} \vee B^*]) \vee (L \wedge \forall x [U \vee B^*])
\end{array}$$

where $L$ is a box in the free variable space. and $U$ is a box-set in the bound variable space. We rewrite it again to make the recursion even more explicit:

$$\begin{array}{rl}
& L \wedge \forall x [U \vee B] \\
\Longleftrightarrow & L^* \wedge \forall x [U^* \vee B^*] \quad \vee \quad L \wedge \forall x [U \vee B^*]
\end{array}$$

where $L^* \equiv L \wedge \underline{B^1}$ and $U^* \equiv U \vee \overline{B^1}$. The recursion terminates since the number of boxes in $B$ decreases.

Now we have the following obvious recursive algorithm. For efficiency, we do some easy check in the step (1) in order to detect trivial cases for which the recursion is not necessary.

**Algorithm 7**

---

$$D \longleftarrow UnivQE(x, B, L, U)$$

*Input*: $x$ is the vector of variables.
$B$ is a box-set.
$L$ is a box in the free variable space.
$U$ is a box-set in the bound variable space.
*Output*: $D$ is a box-set in the free variable space such that $D \Longleftrightarrow L \wedge \forall x[U \vee B]$

(1) [Prune/Recursion base]
If $L$ is empty then set $D$ to be empty and return
If $U$ is the whole bound variable space then set $D$ to be $L$ and return
If $B$ is empty then set $D$ to be empty and return

(2) [Recursion]
Choose a box $B^1$ from $B$
Let $B^*$ be the box-set of the remaining boxes.
$L^* \longleftarrow L \wedge \underline{B^1}$
$U^* \longleftarrow U \vee \overline{B^1}$
$D \longleftarrow UnivQE(x, B^*, L^*, U^*) \vee UnivQE(x, B^*, L, U)$ ■

---

On the top-level, this algorithm is called with $L$ being the whole free-variable space and $U$ being empty, so that the output is a box-set equivalent to $\forall x B$. Some optimization can be done.

- First, the line in Step (2)
$$U^* \longleftarrow U \vee \overline{B^1}$$
can be replaced by the easier operation
$$U^* \longleftarrow U + \overline{B^1}$$
even though $U$ and $\overline{B^1}$ might overlap.
The reason is as follows. Whenever $U$ and $\overline{B^1}$ overlap, $L$ and $\underline{B^1}$ are disjoint (making $L^*$ empty), and thus, the recursive call will not use $U^*$. So, it is safe to carry out the "illegal" operation.

- Next, the line in Step (2)
$$D \longleftarrow UnivQE(x, B^*, L^*, U^*) \vee UnivQE(x, B^*, L, U)$$
can be replaced by the easier one
$$D \longleftarrow UnivQE(x, B^*, L^*, U^*) + UnivQE(x, B^*, L, U)$$
It is because the outputs of the two calls to *UnivQE* are disjoint, as we show now. When $U$ and $\overline{B^1}$ overlap, $L$ and $\underline{B^1}$ are disjoint, and thus the output of the first call is empty. So the disjunction can be trivially replaced by $+$. Thus, now assume that $U$ and $\overline{B^1}$ do not overlap. Let us assume the two outputs overlap, then there exists a point, say $v$, which satisfies both outputs. Then, $v \in \underline{B^1}$. Let $w \in \overline{B^1}$. Then $(v, w) \in B^1$. Since $B^1$ is disjoint from $U$ and $B^*$, $(v, w)$ is not in $U \vee B^*$. This contradicts the fact that $\forall x[U \vee B^*]$ is true on $v$. Thus, the two outputs do not overlap.

### Termination

Does the algorithm (*VolumeApproxQE*) terminate always? The answer is *almost always*. Now we will discuss a few causes of non-termination and solutions for them.

**Non-termination 1**

Assume that the solution set of a formula $F(x, y)$ is $\{(x, y)|0 \leq x - y \leq 1\}$ By plotting the solution set and trying to approximate it with a finite number of boxes, one will discover that there is no way to make the volume-error finite. In general, the method might not terminate if the solution set is un-bounded.

We can avoid such difficulty if we restrict the input to a box-set (finite boxes). This can be done by modifying the algorithm *VolumeApproxQE* so that it has one more input argument, say a box-set $B$, and that it initializes $B_u$ to be $B$. The outputs will be necessary/sufficient conditions for $F \wedge B$.

For the same reason, we also restrict the range of the bound variables. Thus, instead of $\forall x$, we have $\forall x \in C$ where $C$ is a finite box.

For the same reason, we also restrict the range of the bound variables. Thus, instead of $\forall x$, we have $\forall x \in C$ where $C$ is a finite box. Likewise, instead of $\exists x$, we have $\exists x \in C$. One can easily make necessary modifications on the algorithms.

**Non-termination 2** There is one more cause for non-termination. Consider the formula $F$ and the initial box $B$:

$$\begin{array}{lcl} F & : & \forall y \in [0, 1]\ x \neq y \\ B & : & [0, 1] \end{array}$$

By plotting the solution set, one will discover that the volume-difference will not shrink, no matter how accurately the atomic formula $x \neq y$ is approximated. Thus, the algorithm will loop forever. We can avoid such difficulty in several ways.

- In $\forall x F(x, y)$, we restrict that the solution set (in the $x$-space) of $F$ is a closed set for all values of $y$. Likewise in $\exists x F(x, y)$, we restrict that the solution set (in the $x$-space) of $F$ is an open set for all values of $y$. Then, the algorithm will terminate.
- Replace the quantifiers with the generic quantifiers (as defined the section on Generic Quantifier Elim- ination) and use minimal root separation theorems to decide when to abort looping.
- Replace the quantifiers with the measure-quantifier $\mathcal{M}^p$, defined as the sentence

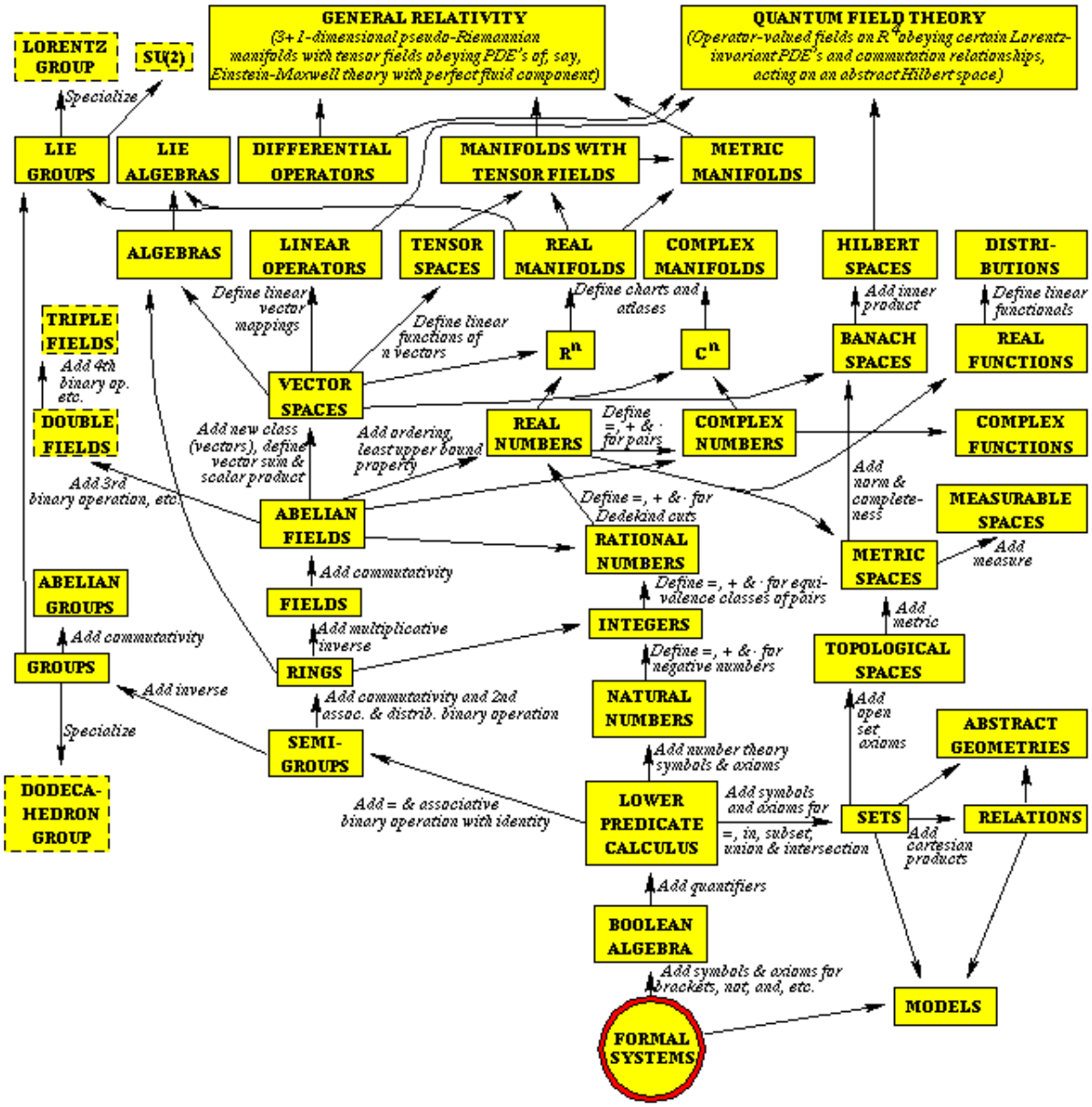
  $$\mathcal{M}^p x \in C\ F(x)$$

  means that $V(F)/V(C) \geq p$. If $0 < p < 1$, then the algorithm terminates.
- Use locally more powerful methods such as: resultants, quadratic forms, Krawszyk's operator, etc

Among them, the method of the measure-quantifier seems to be the most useful in practice.

# Chapter 15

# Potential Future Algebra



# Chapter 16

# Groebner Basis by Siddharth Bhat

Quoting Philip Zucker [Zuck19], the Groebner basis algorithm is:

> The algorithm churns on a set of multivariate polynomials and spits out a new set that is equivalent in the sense that the new set is equal to zero if and only if the original set was. However, now (if you ask for the appropriate term ordering) the polynomials are organized in such a way that they have an increasing number of variables in them. So you solve the 1-variables equations (easy), and substitute into the 2-variable equation. Then that is a 1-variable equation, which you solve (easy) and then you substitute into the three variable equation, and so on. It's analogous to gaussian elimination.

Siddharth Bhat [Bhat19] wrote a couple blog posts on groebner basis which we quote here.

## 16.1 A Grobner Basis example

Here's a fun little problem, whose only solution I know involves a fair bit of math and computer algebra.

We are given the grammar for a language $L$:

```
E = T +_mod8 T | T = -_mod8 T
T = V | V ^ V | V ^ V ^ V
V = 'a1' | 'a2' | ...
```

where `+_mod8` is addition modulo 8, `-_mod8` is subtraction modulo 8, and `^` is XOR.

This language is equipped with the obvious evaluation rules, corresponding to those of arithmetic. We are guaranteed that during evaluation, the variables `a_i` will only have values 0 and 1. Since we have addition, we can perform multiplication by a constant by repeated addition. So we can perform `3*a` as `a+a+a`.

We are then given the input expression

```
(a0 ^ a1 ^ a2 ^ a3)
```

We wish t find an equivalent expression in terms of the above language $L$.

We think of $E$ as some set of logic gates we are allowed to use, and we are trying to express the above operation in terms of these gates.

The first idea that I thought was that of employing a grobner basis, since they essentially embody rewrite rules modulo polynomial equalities, which is precisely our setting here.

In this blog post, I'm going to describe what a grobner basis is and why it's natural to reach for them to solve this problem, the code, and eventual solution.

As a spoiler, the solution is:

```
a^b^c^d =
   -a - b + c + 3*d - 3*axorb - axorc
   + axord - bxorc + bxord + 3*cxord
   - 3*axorbxorc - axorbxord
   + axorcxord + bxorcxord
```

Clearly, this contains only additions/subtractions and multiplication by a constant.

### 16.1.1 What the hell is Grobner Basis?

The nutshell is that a grobner basis is a way to construct rewrite rules which also understand arithmetic (I learnt this viewpoint from the book "Term Rewriting and All That". Fantastic book in general). Expanding on the nutshell, assuming we have a term rewriting system:

```
A -> -1*B -- (1)
C -> B^2  -- (2)
```

over an alphabet `(A, B, C)`.

Now, given the string `C + AB`, we wish to find out if it can be rewritten to 0 or not. Let's try to substitute and see what happens.

```
C + AB -2-> B^2 + AB -1-> B^2 + (-1*B)B
```

At this point, we're stuck! We don't have rewrite rules to allow us to rewrite `(-1*B)B` into `-B^2`. Indeed, creating such a list would be infinitely long. But if we are willing to accept that we somehow have the rewrite rules that correspond to polynomial arithmetic, where we view `A,B,C` as variables, then we can rewrite the above string to 0.

```
B^2 + (-1*B)B -> B^2 - B^2 -> 0
```

A Grobner basis is the algorithmic / mathematical machine that allows us to perform this kind of sumbstitution.

In this example, this might appear stupid; what is so special? We simply substituted variables and arrived at 0 by using arithmetic. What's so complicated about that? To understand why this is not always so easy, let's consider a pathologicial, specially constructed example.

### 16.1.2 A complicated example that shattters dreams

Here's the patheologicial example:

```
A -> 1     -- (1)
AB -> -B^2 -- (2)
```

And we consider the string `S = AB + B^2`. If we blindly apply the first rule, we arrive at

```
S = AB + B^2 -1-> 1B + B^2 = B + B^2 (STUCK)
```

However, if we apply (2) and then (1)

```
S = AB + B^2 -2-> -B^2 + B^2 -> 0
```

This tells us that we can't just apply the rewrite rules willy-nilly. It's sensitive to the order of the rewrites! That is, the rewrite system is not **confluent**.

The grobner basis is a function from rewrite systems to rewrite systems. When given a rewrite system $R$, it produces a new rewrite system $R'$ that is **confluent**. So, we can apply the rewrite rules of $R'$ in any order, and we are guaranteed that we will only get a 0 from $R'$ if and only if we could have gotten a 0 from $R$ for all strings.

We can then go on to phrase this whole rewriting setup in the language of ideals from ring theory, and that is the language in which it is most often described.

Now that we have a handle on what a grobner basis is, let's go on to solve the original problem.

### 16.1.3 An explanation through a slightly simpler problem

I'll first demonstrate the idea of how to solve the original problem by solving a slightly simpler problem:

Rewrite `a^b^c` in terms of `a^b`, `b^c`, `c^a` and the same `+_mod8` instruction set as the original problem. The only difference this time is that we do not have

```
T -> V ^ V ^ V
```

The idea is to construct the polynomial ring over `Z/8Z` (integers modulo 8) with variables a, b, c, axorb, bxorc, axorc. Now, we know that

```
a^b = a + b - 2ab
```

So, we setup rewrite rules such that

```
a + b - 2ab -> axorb
b + c - 2bc -> bxorc
c + a - 2ca -> cxora
```

We construct the polynomial

```
f(a, b, c) = a^b^c
```

which has been written in terms of addition and multiplication, defined as

```
f_orig(a, b, c) = 4*a*b*c - 2*a*b - 2*a*c - 2*b*c + a + b + c
```

We then rewrite `f_orig` with respect to our rewrite rules. Hopefully, the rewrite rules should give us a clean expression in terms of one variable and two-variable xor's. There is the danger that we may have some term such as `a * bxorc`, and we do get such a term `(2*b*axorc)` in this case, but it does not appear in the original problem.

Create a ring with variables a, b, c, axorb, bxorc, axorc.

```
R = IntegerModRing(8)['a, b, c, axorb, bxorc, axorc']
(a, b, c, axorb, bxorc, axorc) = R.gens()
```

XOR in terms of polynomials

```
def xor2(x, y): return x + y - 2*x*y
```

Define the ideal which allows us to rewrite `xor2(a,b)->axorb`, and so on we also add the relation `a^2-a=0 => a=0` or `a=1` in case this helps the solver.

```
I = ideal((axorb - xor2(a,b),
           bxorc - xors(b,c),
           axorc - xor2(a,c),
           a*a-a, b*b-b, c*c-c))
```

The polynomial representing `a^b^c` we wish to reduce

```
f_orig = xor2(a, b, c)
```

We take the groebner basis of the ring to reduce the polynomial $f$

```
IG = I.groebner_basis()
```

We reduce `a^b^c` with respect to the groebner basis.

```
f_reduced = f_orig.reduce(IG)

print("value of a^b^c:\n\t%s\n\treduced: %s" % (f_orig, f_reduced))
```

Code to evaluate the function $f$ on all inputs to check correctness.

```
def evalxor2(f)
  for (i, j, k) in [(i, j, k)
    for i in [0, 1]
      for j in [0, 1]
        for k in [0, 1]]:
          ref = i^^j^^k
          eval = f.substitute(a=i, b=j,
                              axorb=i^^j, bxorc=j^^k, axorc=i^^k)
          print("%s^%s^%s: ref(%s) =?= f(%s): %s" %
            (i, j, k, ref, eval, ref == eval))
```

Check original formulation is correct

```
print("evaluating original f for sanity check")
evalxor2(f_orig)
```

Check reduced formulation is correct

```
print("evaluating reduced f:")
evalxor2(f_reduced)
```

Running the code gives us the redued polynomial `-2*b*axorc+b+axorc` which unfortunately contains a term that is `b*axorc`. So, this approach does not work, and I was informed by my friend that she is unaware of a solution to this problem (writing `a^b^c` in terms of smaller xors and sums).

The full code output is:

```
value of a^b^c:
  4*a*b*c - 2*a*b - 2*a*c - 2*b*c + a + b + c
  reduced: -2*b*axorc + b + axorc
evaluating original f for sanity check:

0^0^0: ref(0) =?= f(0): True
0^0^1: ref(1) =?= f(1): True
0^1^0: ref(1) =?= f(1): True
0^1^1: ref(0) =?= f(0): True
1^0^0: ref(1) =?= f(1): True
1^0^1: ref(0) =?= f(0): True
1^1^0: ref(0) =?= f(0): True
1^1^1: ref(1) =?= f(1): True

evaluating reduced f:
```

```
0^0^0: ref(0) =?= f(0): True
0^0^1: ref(1) =?= f(1): True
0^1^0: ref(1) =?= f(1): True
0^1^1: ref(0) =?= f(0): True
1^0^0: ref(1) =?= f(1): True
1^0^1: ref(0) =?= f(0): True
1^1^0: ref(0) =?= f(0): True
1^1^1: ref(1) =?= f(1): True
```

That is, both the original polynomial and the reduced polynomial match the expected results. But the reduced polynomial is not in our language $L$, since it has a term that is a product of $b$ with *axorc*.

```
-a - b + c + 3*d - 3*axorb - axorc
+ axord - bxorc + bxord + 3*cxord
- 3*axorbxorc - axorbxord
+ axorcxord + bxorcxord
```

which happily has no products between terms! It also passes our sanity check, so we've now found the answer.

The full output is:

```
value of a^b^c^d:
  4*a*b*c + 4*a*b*d + 4*a*c*d + 4*b*c*d - 2*a*b -2*a*c - 2*b*c
  - 2*a*d - 2*b*d - 2*c*d + a + b + c + d
 reduced:
   -a - b + c + 3*d - 3*axorb - axorc + axord - bxorc + bxord
   + 3*cxord - 3*axorbxorc - axorbxord + axorcxord + bxorcxord


evaluating original a^b^c^d
0^0^0^0: ref(0) =?= f(0): True
0^0^0^1: ref(1) =?= f(1): True
0^0^1^0: ref(1) =?= f(1): True
0^0^1^1: ref(0) =?= f(0): True
0^1^0^0: ref(1) =?= f(1): True
0^1^0^1: ref(0) =?= f(0): True
0^1^1^0: ref(0) =?= f(0): True
0^1^1^1: ref(1) =?= f(1): True
1^0^0^0: ref(1) =?= f(1): True
1^0^0^1: ref(0) =?= f(0): True
1^0^1^0: ref(0) =?= f(0): True
1^0^1^1: ref(1) =?= f(1): True
1^1^0^0: ref(0) =?= f(0): True
1^1^0^1: ref(1) =?= f(1): True
1^1^1^0: ref(1) =?= f(1): True
1^1^1^1: ref(0) =?= f(0): True

evaluating reduced a^b^c^d
0^0^0^0: ref(0) =?= f(0): True
0^0^0^1: ref(1) =?= f(1): True
0^0^1^0: ref(1) =?= f(1): True
0^0^1^1: ref(0) =?= f(0): True
0^1^0^0: ref(1) =?= f(1): True
0^1^0^1: ref(0) =?= f(0): True
0^1^1^0: ref(0) =?= f(0): True
```

```
   0^1^1^1: ref(1) =?= f(1): True
   1^0^0^0: ref(1) =?= f(1): True
   1^0^0^1: ref(0) =?= f(0): True
   1^0^1^0: ref(0) =?= f(0): True
   1^0^1^1: ref(1) =?= f(1): True
   1^1^0^0: ref(0) =?= f(0): True
   1^1^0^1: ref(1) =?= f(1): True
   1^1^1^0: ref(1) =?= f(1): True
   1^1^1^1: ref(0) =?= f(0): True
```

Code for `a^b^c^d`

```
def xor3(x, y, z): return xor2(x, xor2(y, z))

R = IntegerModRing(8)['a, b, c, d, axorb, axorc, axord,
                       bxorc, bxord, cxord, axorbxorc,
                       axorbxord, axorcxord, bxorcxord']
(a, b, c, d, axorb, axorc, axord, bxorc, bxord, cxord,
  axorbxorc, axorbxord, axorcxord, bxorcxord) = R.gens()
I = ideal((axorb - xor2(a, b),
           axorc - xor2(a, c),
           axord - xor2(a, d),
           bxorc - xor2(b, c),
           bxord - xor2(b, d),
           cxord - xor2(c, d),
           axorbxorc - xor3(a, b, c),
           axorbxord - xor3(a, b, d),
           axorcxord - xor3(a, c, d),
           bxorcxord - xor3(b, c, d),
           a*a-a,
           b*b-b,
           c*c-c,
           d*d-d))
IG = I.groebner_basis()
f_orig = (xor2(a, xor2(b, xor2(c, d))))
f_reduced = f_orig.reduce(IG)
print("value of a^b^c^d:\n\t%s\n\treduced: %s" % (f_orig, f_reduced))

def evalxor3(f):
  for (i, j, k, l) in [(i, j, k, l)
    for i in [0, 1]
      for j in [0, 1]
        for k in [0, 1]
          for l in [0, 1]]:
    ref = i^^j^^k^^l
    eval = f.substitute(a=i, b=j, c=k, d=l,
                        axorb=i^^j, axorc=i^^k, axord=i^^l,
                        bxorc=j^^k, bxord=j^^l,
                        cxord=k^^l, axorbxorc=i^^j^^k,
                        axorbxord=i^^j^^l, axorcxord=i^^k^^l,
                        bxorcxord=j^^k^^l)
    print("%s^%s^%s^%s: ref(%s) =?= f(%s): %s" %
          (i, j, k, l, ref, eval, ref == eval))

print("evaluating original a^b^c^d")
```

```
evalxor3(f_orig)

print("evaluating reduced a^b^c^d")
evalxor3(f_reduced)
```

## 16.2 Ideals as Rewrite Systems

### 16.2.1 A motivating example

The question a Grobner basis allows us to answer is this: can the polynomial

$$p(x, y) = xy^2 + y$$

be factorized in terms of

$$a(x, y) = xy + 1, \quad b(x, y) = y^2 - 1$$

such that

$$p(x, y) = f(x, y)a(x, y) + g(x, y)b(x, y)$$

for some *arbitrary* polynomials

$$f(x, y), g(x, y) \in R[x, y]$$

One might imagine, "well, I'll divide and see what happens!". Now, there are two routes to do down:

$$xy^2 + y = y(xy + 1) = ya(x, y) + 0b(x, y)$$

Well, problem solved?

$$xy^2 + y = xy^2 - x + x + y = x(y^2 - 1) + x + y = xb(x, y) + (x + y)$$

Now what? We're stuck, and we can't apply $a(x, y)$.

So, clearly, the *order* in which we perform the factorization / division starts to matter! Ideally, we want an algorithm which is *not sensitive* to the order in which we choose to apply these changes. $x^2 + 1$

### 16.2.2 The rewrite rule perspective

An alternative viewpoint of asking "can this be factorized", is to ask "can we look at the factorization as a rewrite rule?". For this perspective, notice that "factorizing" in terms of $xy + 1$ is the same as being able to set $xy = -1$, and then have the polynomial collapse to zero. For the more algebraic minded, this relates to the fact that

$$R[x]/p(x) \approx R(\text{roots of p})$$

The intuition behind this is that when we "divide by $xy + 1$", really what we are doing is we are setting $xy + 1 = 0$, and then seeing what remains. But

$$xy + 1 = 0 \Leftrightarrow xy = -1$$

. Thus, we can look at the original question as:

How can we apply the rewrite rules $xy \to -1$, $y^2 \to 1$, along with the regular rewrite rules of polynomial arithmetic to the polynomial $p(x, y) = xy^2 + y$, such that we end with the value 0?

Our two derivations above correspond to the application of the rules:

$$xy + y \xrightarrow{xy=-1} -y + y = 0$$

$$xy^2 + y \xrightarrow{y^2=1} x + y \not\to \text{stuck!}$$

That is, our rewrite rules are not confluent.

The grobner basis is a mathematical object, which is a *confluent set of rewrite rules* for the above problem. That is, it's a set of polynomials which manage to find the rewrite

$$p(x, y) \xrightarrow{*} 0$$

regardless of the order in which we apply them. It's also *correct*, in that it only rewrites to 0 if the original system has *some way* to rewrite to 0.

### 16.2.3 Buchberger's algorithm

We need to identify *critical pairs*, which in this setting are called S-polynomials.

Let

$$f_i = H(f_i) + R(f_i)$$
$$f_j = H(f_j) + R(f_j)$$

.

$$m = lcm(H(f_i), H(f_j))$$

and let $m_i$, $m_j$ be monomials such that

$$m_i * H(f_i) = m = m_j * H(f_j)$$

The S-polynomial induced by $f_i$, $f_j$ is defined as

$$S(f_i, f_j) = m_i f_i - m_i f_j$$

# Chapter 17

# Greatest Common Divisor

Greatest Common Divisor

# Chapter 18

# Polynomial Factorization

Polynomial Factorization

# Chapter 19

# Differential Forms

This is quoted from Wheeler [Whee12].

## 19.1 From differentials to differential forms

In a formal sense, we may define differentials as the vector space of linear mappings from curves to the reals, that is, given a differential $df$ we may use it to map any curve, C $\in C$ to a real number simply by integrating:

$$df : C \to R$$

$$x = \int_C df$$

This suggests a generalization, since we know how to integrate over surfaces and volumes as well as curves. In higher dimensions we also have higher order multiple integrals. We now consider the integrands of arbitrary multiple integrals

$$\int f(x)dl, \quad \int\int f(x)dS, \quad \int\int\int f(x)dV$$

Much of their importance lies in the coordinate invariance of the resulting integrals.

One of the important properties of integrands is that they can all be regarded as oriented. If we integrate a line integral along a curve from $A$ to $B$ we get a number, while if we integrate from $B$ to $A$ we get minus the same number,

$$\int_A^B f(x)dl = -\int_B^A f(x)dl$$

We can also demand oriented surface integrals, so the surface integral

$$\int\int \mathbf{A} \cdot \mathbf{n}\, dS$$

changes sign if we reverse the direction of the normal to the surface. This normal can be thought of as the cross product of two basis vectors within the surface. If these basis vectors' cross product is taken in one order, **n** has one sign. If the opposite order is taken then **-n** results. Similarly, volume integrals change sign if we change from a right- or left-handed coordinate system.

### 19.1.1 The wedge product

We can build this alternating sign into our convention for writing differential forms by introducing a formal antisymmetric product, called the *wedge* product, symbolized by $\wedge$, which is defined to give these differential elements the proper signs. Thus, surface integrals will be written as integrals over the products

$$\mathbf{dx} \wedge \mathbf{dy}, \mathbf{dy} \wedge \mathbf{dz}, \mathbf{dz} \wedge \mathbf{dx}$$

with the convention that $\wedge$ is antisymmetric:

$$\mathbf{dx} \wedge \mathbf{dy} = -\mathbf{dy} \wedge \mathbf{dx}$$

under the interchange of any two basis forms. This automatically gives the right orientation of the surface. Similarly, the volume element becomes

$$\mathbf{V} = \mathbf{dx} \wedge \mathbf{dy} \wedge \mathbf{dz}$$

which changes sign if any pair of the basis elements are switched.

We can go further than this by formalizing the full integrand. For a line integral, the general form of the integrand is a linear combination of the basis differentials,

$$\mathbf{A}_x \mathbf{dx} + \mathbf{A}_y \mathbf{dy} + \mathbf{A}_z \mathbf{dz}$$

Notice that we simply add the different parts. Similary, a general surface integrand is

$$\mathbf{A}_z \mathbf{dx} \wedge \mathbf{dy} + \mathbf{A}_y \mathbf{dz} \wedge \mathbf{dx} + \mathbf{A}_x \mathbf{dy} \wedge \mathbf{dz}$$

while the volume integrand is

$$f(x)\ \mathbf{dx} \wedge \mathbf{dy} \wedge \mathbf{dz}$$

These objects are called *differential forms*.

Clearly, differential forms come in severaly types. Functions are called 0-forms, line elements 1-forms, surface elements 2-forms, and volume elements are called 3-forms. These are all the types that exist in 3-dimensions, but in more than three dimensions we can have $p$-forms with $p$ ranging from zero to the dimension, $d$, of the space. Since we can take arbitrary linear combinations of $p$-forms, they form a vector space, $\Lambda_p$.

We can always wedge together any two forms. We assume this wedge product is associative, and obeys the usual distributive laws. The wedge product of a $p$-form with a $q$-form is a $(p+q)$-form.

Notice that the antisymmetry is all we need to rearrange any combination of forms. In general, wedge products of even order forms with any other forms commute while wedge products of pairs of odd-order forms anticommute. In particular, functions (0-forms) commute with all $p$-forms. Using this, we may interchange the order of a line element and a surface area, for if

$$\mathbf{l} = A\ \mathbf{dx}$$
$$\mathbf{S} = B\ \mathbf{dy} \wedge \mathbf{dz}$$

then

$$\begin{aligned}
\mathbf{l} \wedge \mathbf{S} &= (A\ \mathbf{dx}) \wedge (B\ \mathbf{dy} \wedge \mathbf{dz}) \\
&= A\ \mathbf{dx} \wedge B\ \mathbf{dy} \wedge \mathbf{dz} \\
&= AB\ \mathbf{dx} \wedge \mathbf{dy} \wedge \mathbf{dz} \\
&= -AB\ \mathbf{dy} \wedge \mathbf{dx} \wedge \mathbf{dz} \\
&= AB\ \mathbf{dy} \wedge \mathbf{dz} \wedge \mathbf{dx} \\
&= \mathbf{S} \wedge \mathbf{l}
\end{aligned}$$

but the wedge product of two line elements changes sign, for if

$$\mathbf{l}_1 = A\ \mathbf{dx}$$

$$\mathbf{l}_2 = B\ \mathbf{dy} + C\ \mathbf{dz}$$

then

$$\begin{aligned}
\mathbf{l}_1 \wedge \mathbf{l}_2 &= (A\ \mathbf{dx}) \wedge (B\ \mathbf{dy} + C\ \mathbf{dz}) \\
&= A\ \mathbf{dx} \wedge B\ \mathbf{dy} + A\ \mathbf{dx} \wedge C\ \mathbf{dz} \\
&+ AB\ \mathbf{dx} \wedge \mathbf{dy} + AC\ \mathbf{dx} \wedge \mathbf{dz} \\
&= -AB\ \mathbf{dy} \wedge \mathbf{dx} - AC\ \mathbf{dz} \wedge \mathbf{dz} \\
&= -B\ \mathbf{dy} \wedge A\ \mathbf{dx} - C\ \mathbf{dz} \wedge A\ \mathbf{dx} \\
&= -\mathbf{l}_2 \wedge \mathbf{l}_1
\end{aligned}$$

For any odd-order form, $\omega$, we immediately have

$$\omega \wedge \omega = -\omega \wedge \omega = 0$$

In 3-dimensions there are no 4-forms because anything we try to construct must contain a repeated basis form. For example,

$$\begin{aligned}
\mathbf{l} \wedge \mathbf{V} &= (A\ \mathbf{dx}) \wedge (B\ \mathbf{dx} \wedge \mathbf{dy} \wedge \mathbf{dz}) \\
&= AB\ \mathbf{dx} \wedge \mathbf{dx} \wedge \mathbf{dy} \wedge \mathbf{dz} \\
&= 0
\end{aligned}$$

since $\mathbf{dx} \wedge \mathbf{dx} = 0$. The same occurs for anything we try. Of course, if we have more dimensions then there are more independent directions and we can find nonzero 4-forms. In general, in $d$-dimensions we can find $d$-forms, but no $(d+1)$-forms.

Now suppose we want to change coordinates. How does an integrand change? Suppose Cartesian coordinates (x,y) in the plane are given as some functions of new coordinates (u,v). Then we already know that differentials change according to

$$\mathbf{dx} = \mathbf{dx}(u,v) = \frac{\partial x}{\partial u}\mathbf{du} + \frac{\partial x}{\partial v}\mathbf{dv}$$

and similarly for $\mathbf{dy}$, applying the usual rules for partial differentiation. Notice what happens when we use the wedge product to calculate the new area element:

$$\begin{aligned}
\mathbf{dx} \wedge \mathbf{dy} &= \left(\frac{\partial x}{\partial u}\mathbf{du} + \frac{\partial x}{\partial v}\mathbf{dv}\right) \wedge \left(\frac{\partial y}{\partial u}\mathbf{du} + \frac{\partial y}{\partial v}\mathbf{dv}\right) \\
&= \frac{\partial x}{\partial v}\frac{\partial y}{\partial u}\mathbf{dv} \wedge \mathbf{du} + \frac{\partial x}{\partial u}\frac{\partial y}{\partial v}\mathbf{du} \wedge \mathbf{dv} \\
&= \left(\frac{\partial x}{\partial u}\frac{\partial y}{\partial v} - \frac{\partial x}{\partial v}\frac{\partial y}{\partial u}\right)\mathbf{du} \wedge \mathbf{dv} \\
&= J\ \mathbf{du} \wedge \mathbf{dv}
\end{aligned}$$

where

$$J = \det \begin{pmatrix} \dfrac{\partial x}{\partial u} & \dfrac{\partial x}{\partial v} \\ \dfrac{\partial y}{\partial u} & \dfrac{\partial y}{\partial v} \end{pmatrix}$$

is the Jacobian of the coordinate transformation. This is exactly the way that an area element changes when we change coordinates. Notice the Jacobian coming out automatically. We couldn't ask for more - the wedge product not only gives us the right signs for oriented areas and volumes, but gives us the right transformation to new coordinates. Of course the volume change works, too.

Under a coordinate transformation

$$x \to x(u, v, w)$$

$$y \to y(u, v, w)$$

$$z \to z(u, v, w)$$

the new volume element is the full Jacobian times the new volume form,

$$\mathbf{dx} \wedge \mathbf{dy} \wedge \mathbf{dz} = J(xyz; uvw)\ \mathbf{du} \wedge \mathbf{dv} \wedge \mathbf{dw}$$

So the wedge product successfully keesp track of $p$-dim volumes and their orientations in a coordinate invariant way. Now any time we have an integral, we can regard the integrand as being a differential form. But all of this can go much further. Recall our proof that 1-forms form a vector space. Thus, the differential, $\mathbf{dx}$ of $x(u, v)$ given above is just a gradient. It vanishes along surfaces where $x$ is constant, and the components of the vector

$$\left(\frac{\partial x}{\partial u}, \frac{\partial x}{\partial v}\right)$$

point in a direction normal to those surfaces. So symbols like $\mathbf{dx}$ or $\mathbf{du}$ contain directional information. Writing them with a boldface $\mathbf{d}$ indicates this vector character. Thus, we write

$$\mathbf{A} = A_i \mathbf{dx^i}$$

Let

$$f(x, y) = axy$$

The vector with components

$$\left(\frac{\partial f}{\partial u}, \frac{\partial f}{\partial v}\right)$$

is perpendicular to the surfaces of constant $f$.

We have defined forms, have written down their formal properties, and have used those properties to write them in components. Then, we define the wedge product, which enables us to write $p$-dimensional integrands as $p$-forms in such a way that the orientation and coordinate transformation properties of the integrals emerges automatically.

Though it is 1-forms, $A_i \mathbf{dx^i}$ that corresponding to vectors, we have defined a product of basis forms that we can generalize to more complicated objects. Many of these objects are

already familiar. Consider the product of two 1-forms.

$$\begin{aligned}
\mathbf{A}\wedge\mathbf{B} &= A_i\ \mathbf{dx}^i\wedge B_j\ \mathbf{dx}^j\\
&= A_iB_j\ \mathbf{dx}^i\wedge\mathbf{dx}^j\\
&= \frac{1}{2}A_iB_j\ (\mathbf{dx}^i\wedge\mathbf{dx}^j-\mathbf{dx}^j\wedge\mathbf{dx}^i)\\
&= \frac{1}{2}(A_iB_j\ \mathbf{dx}^i\wedge\mathbf{dx}^j-A_iB_j\ \mathbf{dx}^j\wedge\mathbf{dx}^i)\\
&= \frac{1}{2}(A_iB_j\ \mathbf{dx}^i\wedge\mathbf{dx}^j-A_jB_i\ \mathbf{dx}^i\wedge\mathbf{dx}^j)\\
&= \frac{1}{2}(A_iB_j-A_jB_i)\ \mathbf{dx}^i\wedge\mathbf{dx}^j
\end{aligned}$$

The coefficients

$$A_iB_j-A_jB_i$$

are essentially the components of the cross product. We will see this in more detail below when we discuss the curl.

### 19.1.2 The exterior derivative

We may regard the differential of any function, say $f(x,y,z)$, as the 1-form:

$$\begin{aligned}
\mathbf{d}f &= \frac{\partial f}{\partial x}\mathbf{d}x+\frac{\partial f}{\partial y}\mathbf{d}y+\frac{\partial f}{\partial z}\mathbf{d}z\\
&= \frac{\partial f}{\partial x^i}\mathbf{d}x^i
\end{aligned}$$

Since a fnction is a 0-form then we can imagine an operator $\mathbf{d}$ that differentiates any 0-form to give a 1-form. In Cartesian coordinates, the coefficients of this 1-form are just the Cartesian components of the gradient.

The operator $\mathbf{d}$ is called the *exterior derivative*, and we may apply it to any $p$-form to get a $(p+1)$-form. The extension is defined as follows. First consider a 1-form

$$\mathbf{A}=A_i\ \mathbf{dx}^i$$

We define

$$\mathbf{dA}=\mathbf{d}A_i\wedge\mathbf{dx}^i$$

Similarly, since an arbitrary $p$-form in $n$-dimensions may be written as

$$\omega=A_{i_1,i_2,\cdots,i_p}\wedge\mathbf{dx}^{i_1}\wedge\mathbf{dx}^{i_2}\cdots\wedge\mathbf{dx}^{i_p}$$

we define the exterior derivative of $\omega$ to be a $(p+1)$-form

$$\mathbf{d}\omega=\mathbf{d}A_{i_1,i_2,\cdots,i_p}\wedge\mathbf{dx}^{i_1}\wedge\mathbf{dx}^{i_2}\cdots\wedge\mathbf{dx}^{i_p}$$

Let's see what happens if we apply $\mathbf{d}$ twice to the Cartesian coordinate, $x$ regarded as a function of $x,y$ and $z$:

$$\begin{aligned}
\mathbf{d}^2x &= \mathbf{d}(\mathbf{d}x)\\
&= \mathbf{d}(1\mathbf{d}x)\\
&= \mathbf{d}(1)\wedge\mathbf{d}x\\
&= 0
\end{aligned}$$

since all derivatives of the constant function $f = 1$ are zero. The same applies if we apply $\mathbf{d}$ twice to *any* function:

$$\begin{aligned}
\mathbf{d}^2 f &= \mathbf{d}(\mathbf{d}f) \\
&= \mathbf{d}\left(\frac{\partial f}{\partial x^i}\mathbf{d}x^i\right) \\
&= \mathbf{d}\left(\frac{\partial f}{\partial x^i} \wedge \mathbf{d}x^i\right) \\
&= \left(\frac{\partial^2 f}{\partial x^j \partial x^i}\mathbf{d}x^j\right) \wedge \mathbf{d}x^i \\
&= \frac{\partial^2 f}{\partial x^j \partial x^i}\mathbf{d}x^j \wedge \mathbf{d}x^i
\end{aligned}$$

By the same argument we used to get the components of the curl, we may write this as

$$\begin{aligned}
\mathbf{d}^2 f &= \frac{1}{2}\left(\frac{\partial^2 f}{\partial x^j \partial x^i} - \frac{\partial^2 f}{\partial x^i \partial x^j}\right)\mathbf{d}x^j \wedge \mathbf{d}x^i \\
&= 0
\end{aligned}$$

since partial derivatives commute.

Poincaré Lemma: $\mathbf{d}^2\omega = 0$ where $\omega$ is an arbitrary $p$-form.

Next, consider the effect on $\mathbf{d}$ on an arbitrary 1-form. We have

$$\begin{aligned}
\mathbf{dA} &= \mathbf{d}(A_i \mathbf{d}x^i) \\
&= \left(\frac{\partial A_i}{\partial x^j}\mathbf{d}x^j\right) \wedge \mathbf{d}x^i \\
&= \frac{1}{2}\left(\frac{\partial A_i}{\partial x^j} - \frac{\partial A_j}{\partial x^i}\right)\mathbf{d}x^j \wedge \mathbf{d}x^i
\end{aligned}$$

We have the components of the curl of the vector $\mathbf{A}$. We must be careful here, however, because these are the components of the curl only in Cartesian coordinates. Later we will see how these components relate to those in a general coordinate system. Also, recall that the components $A_i$ are distinct from the usual vector components $A^i$. These differences will be resolved when we give a detailed discussion of the metric. Ultimately, the action of $\mathbf{d}$ on a 1-form gives us a coordinate invariant way to calculate the curl.

Finally, suppose we have a 2-form expressed as

$$\mathbf{S} = A_z\ \mathbf{d}x \wedge \mathbf{d}y + A_y\ \mathbf{d}z \wedge \mathbf{d}x + A_x\ \mathbf{d}y \wedge \mathbf{d}z$$

Then apply the exterior derivative gives

$$\begin{aligned}
\mathbf{d}S &= \mathbf{d}A_z \wedge \mathbf{d}x \wedge \mathbf{d}y + \mathbf{d}A_y \wedge \mathbf{d}z \wedge \mathbf{d}x + \mathbf{d}A_x \wedge \mathbf{d}y \wedge \mathbf{d}z \\
&= \frac{\partial A_z}{\partial z}\mathbf{d}z \wedge \mathbf{d}x \wedge \mathbf{d}y + \frac{\partial A_y}{\partial y}\mathbf{d}y \wedge \mathbf{d}z \wedge \mathbf{d}x + \frac{\partial A_x}{\partial x}\mathbf{d}x \wedge \mathbf{d}y \wedge \mathbf{d}z \\
&= \left(\frac{\partial A_z}{\partial z} + \frac{\partial A_y}{\partial y} + \frac{\partial A_x}{\partial x}\right)\ \mathbf{d}x \wedge \mathbf{d}y \wedge \mathbf{d}z
\end{aligned}$$

so that the exterior derivative can also reproduce the divergence.

### 19.1.3 The Hodge dual

To truly have the curl we need a way to turn a 2-form into a vector, i.e., a 1-form and a way to turn a 3-form into a 0-form. This leads us to introduce the Hodge dual , or star, operator $\star$.

Notice that in 3-dim, both 1-forms and 2-forms have three independent components, while both 0- and 3-forms have one component. This suggests that we can define an invertible mapping between these pairs. In Cartesian coordinates, suppose we set

$$\begin{aligned} \star(\mathbf{dx}\wedge\mathbf{dy}) &= \mathbf{dz} \\ \star(\mathbf{dy}\wedge\mathbf{dz}) &= \mathbf{dx} \\ \star(\mathbf{dz}\wedge\mathbf{dx}) &= \mathbf{dy} \\ \star(\mathbf{dx}\wedge\mathbf{dy}\wedge\mathbf{dz}) &= 1 \end{aligned}$$

and further require that the star be its own inverse

$$\star\star = 1$$

With these rules we can find the Hodge dual of any form in 3-dim.

The dual of the general 1-form

$$\mathbf{A} = A_i\mathbf{dx}^i$$

is the 2-form

$$S = A_z\ \mathbf{dx}\wedge\mathbf{dy} + A_y\ \mathbf{dz}\wedge\mathbf{dx} + A_x\ \mathbf{dy}\wedge\mathbf{dz}$$

For an arbitrary (Cartesian) 1-form

$$\mathbf{A} = A_i\mathbf{dx}^i$$

that

$$\star\mathbf{d}\star\mathbf{A} = div\mathbf{A}$$

The curl of $\mathbf{A}$

$$curl(\mathbf{A}) = \left(\frac{\partial A_y}{\partial z} - \frac{\partial A_z}{\partial y}\right)\mathbf{dx} + \left(\frac{\partial A_z}{\partial x} - \frac{\partial A_x}{\partial z}\right)\mathbf{dy}\left(\frac{\partial A_x}{\partial y} - \frac{\partial A_y}{\partial x}\right)\mathbf{dz}$$

Three operations - the wedge product $\wedge$, the exterior derivative $\mathbf{d}$, and the Hodge dual $\star$ - together encompass the usual dot and cross products as well as the divergence, curl and gradient. In fact, they do much more - they extend all of these operations to arbitrary coordinates and arbitrary numbers of dimensions. To explore these generalizations, we must first explore properties of the metric and look at coordinate transformations. This will allow us to define the Hodge dula in arbitrary coordinates.

# Chapter 20

# Pade approximant

Pade approximant

# Chapter 21

# Schwartz-Zippel lemma and testing polynomial identities

Schwartz-Zippel lemma and testing polynomial identities

# Chapter 22

# Chinese Remainder Theorem

Chinese Remainder Theorem

# Chapter 23

# Gaussian Elimination

Gaussian Elimination

# Chapter 24

# Diophantine Equations

Diophantine Equations

# Bibliography

[Abad94] Martin Abadi and Luca Cardelli. A Semantics of Object Types. In *Symp. on Logic in Computer Science '94*. IEEE, 1994.

**Abstract:** We give a semantics for a typed object calculus, an extension of System **F** with object subsumption and method override. We interpret the calculus in a per model, proving the soundness of both typing and equational rules. This semantics suggests a syntactic translation from our calculus into a simpler calculus with neither subtyping nor objects

[Abad94a] Martin Abadi and Luca Cardelli. A Theory of Primitive Objects: Untyped and First-Order Systems. In *Proc. European Symposium on Programming*, 1994.

**Abstract:** We introduce simple object calculi that support method override and object subsumption. We give an untyped calculus, typing rules, and equational rules. We illustrate the expressiveness of our calculi and the pitfalls that we avoid.

[Abda86] S. Kamal Abdali, Guy W. Cherry, and Neil Soiffer. A Smalltalk System for Algebraic Manipulation. In *OOPSLA 86*, pages 277–293, 1986.

**Abstract:** This paper describes the design of an algebra system Views implemented in Smalltalk. Views contains facilities for dynamic creation and manipulation of computational domains, for viewing these domains as various categories such as groups, rings, or fields, and for expressing algorithms generically at the level of categories. The design of Views has resulted in the addition of some new abstractions to Smalltalk that are quite useful in their own right. Parameterized classes provide a means for run-time creation of new classes that exhibit generally very similar behavior, differing only in minor ways that can be described by different instantiations of certain parameters. Categories allow the abstraction of the common behavior of classes that derives from the class objects and operations satisfying certain laws independently of the implementation of those objects and operations. Views allow the run-time association of classes with categories (and of categories with other categories), facilitating the use of code written for categories with quite different interpretations of operations. Together, categories and views provide an additional mechanism for code sharing that is richer than both single and multiple inheritance. The paper gives algebraic as well as non-algebraic examples of the above-mentioned features.

[Abla98] Rafal Ablamowicz. Spinor Representations of Clifford Algebras: A Symbolic Approach. *Computer Physics Communications*, 115(2-3):510–535, December 1998.

[Adax12] ISO/IEC 8652:2012(E). *Ada Reference Manual*. U.S. Government, 2012.

**Link:** http://www.ada-auth.org/standards/12rm/RM-Final.pdf

[Ahox86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986, 978-0201100884.

[Alfo92] W.R. Alford, A. Granville, and C. Pomerance. There are Infinitely Many Carmichael Numbers, 1992.

**Comment:** Preprint

[Altm05] Simon L. Altmann. *Rotations, Quaternions, and Double Groups*. Dover Publications, Inc., 2005, 0-486-44518-6.

[Amad93] Roberto M. Amadio and Luca Cardelli. Subtyping Recursive Types. *TOPLAS '93*, 15(4):575–631, 1993.

**Abstract:** We investigate the interactions of subtyping and recursive types, in a simply typed $\lambda$-calculus. The two fundamental questions here are whether two (recursive)types are in the subtype relation and whether a term has a type. To address the first question, we relate various definitions of type equivalence and subtyping that are induced by a model, an ordering on infinite trees, an algorithm, and a set of type rules. We show soundness and completeness among the rules, the algorithm, and the tree semantics. We also prove soundness and a restricted form of completeness for the model. To address the second question, we show that to every pair of types in the subtype relation we can associate a term whose denotation is the uniquely determined coercion map between the two types. Moreover, we derive an algorithm that, when given a term with implicit coercions, can infer its least type whenever possible.

[Anai00] Hirokazu Anai and Volker Weispfenning. Deciding linear-trigonometric problems. In *Proc ISSAC'00*, pages 14–22. ACM, 2000, 1-58113-218-2.

**Abstract:** In this paper, we present a decision procedure for certain linear-trigonometric problems for the reals and integers formalized in a suitable first-order language. The inputs are restricted to formulas, where all but one of the quantified variables occur linearly and at most one occurs both linearly and in a specific trigonometric function. Moreover we may allow in addition the integer-part operation in formulas. Besides ordinary quantifiers, we allow also counting quantifiers. Furthermore we also determine the qualitative structure of the connected components of the satisfaction set of the mixed linear-trigonometric variable. We also consider the decision of these problems in subfields of the real algebraic numbers.

[Arna91] F. Arnault. Le Test de Primalite de Rabin-Miller: Un Nombre compose qui le "passe", 1991.

**Comment:** Report 61

[Arno81] Dennis Soulé Arnon. *Algorithms for the Geometry of Semi-algebraic Sets*. PhD

thesis, University of Wisconsin-Madison, 1981.

**Abstract:** Let A be a set of polynomials in r variables with integer coefficients. An $A$-invariant cylindrical algebraic decomposition (cad) of $r$-dimensional Euclidean space (G. Collins, Lect. Notes Comp. Sci., 33, Springer-Verlag, 1975, pp 134-183) is a certain cellular decomposition of $r$-space, such that each cell is a semi-algebraic set, the polynomials of $A$ are sign-invariant on each cell, and the cells are arranged into cylinders. The cad algorithm given by Collins provides, among other applications, the fastest known decision procedure for real closed fields, a cellular decomposition algorithm for semi-algebraic sets, and a method of solving nonlinear (polynomial) optimization problems exactly. The time-consuming calculations with real algebraic numbers required by the algorithm have been an obstacle to its implementation and use. The major contribution of this thesis is a new version of the cad algorithm for $r \leq 3$, in which one works with maximal connected $A$-invariant collections of cells, in such a way as to often avoid the most time-consuming algebraic number calculations. Essential to this new cad algorithm is an algorithm we present for determination of adjacenies among the cells of a cad. Computer programs for the cad and adjacency algorithms have been written, providing the first complete implementation of a cad algorithm. Empirical data obtained from application of these programs are presented and analyzed.

[Arno84] Dennis S. Arnon, George E. Collins, and Scott McCallum. Cylindrical Algebraic Decomposition II: An Adjacency Algorithm for the Plane. *SIAM J. Comput.*, 13(4):878–889, 1984.

**Abstract:** Given a set of r-variate integral polynomials, a *cylindrical algebraic decomposition (cad)* of euclidean r-space $E^r$ partitions $E^r$ into connected subsets compaitible with the zeros of the polynomials. Each subset is a *cell*. Informally, two cells of a cad are *adjacent* if they touch each other; formally, they are adjacent if their union is connected. In applications of cad's one often wishes to know the adjacent pairs of cells. Previous algorithms for cad construction (such as that given in Part I of this paper) have not actually determined them. We give here in Part II an algorithm which determines the pairs of adjacent cells as it constructs a cad of $E^2$.

[Arno88a] Dennis Arnon and Bruno Buchberger. Algorithms in Real Algebraic Geometry, 1988.

[Arno88b] Dennis S. Arnon, George E. Collins, and Scott McCallum. An Adjacency algorithm for cylindrical algebraic decompositions of three-dimensional space. *J. Symbolic Computation*, 5(1-2):163–187, 1988.

**Abstract:** Let $A \subset \mathbb{Z}[x_1, \ldots, x_r]$ be a finite set. An *A-invariant cylindrical algebraic decomposition (cad)* is a certain partition of $r$-dimenslonal euclidean space $E^r$ into semi-algebraic cells such that the value of each $A_i \in A$ has constant sign (positive, negative, or zero) throughout each cell. Two cells are adjacent if their union is connected. We give an algorithm that determines the adjacent pairs of

cells as it constructs a cad of $E^3$. The general teehnlque employed for $^3$ adjacency determination is projection into $E^2$, followed by application of an existing $E^2$ adjacency elgorlthm (Arnon, Collins, McCallum, 1984). Our algorithm has the following properties: (1) it requires no coordinate changes, end (2) in any cad of $E^1$, $E^2$, or $E^3$ that it builds, the boundary of each cell is a (disjoint) union of lower-dlmenaionel cells.

[Ashx89] D.W. Ash, I.F. Black, and S.A. Vanstone. Low Complexity Normal Bases. *Discrete Applied Mathematics*, 25:191–210, 1989.

[Bare84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier Science, 1984.

[Bart94] John J. Barton and Lee R. Nackman. *Scientific and Engineering C++*. Pearson, 1994, 97800201533934.

[Baum95] Gerald Baumgartner and Ryan D. Stansifer. A Proposal to Study Type Systems for Computer Algebra. technical report 90-07.0, RISC-LINZ, 1995.

**Abstract:** It is widely recognized that programming languages should offer features to help structure programs. To achieve this goal, languages like Ada , Modula-2 , object-oriented languages, and functional languages have been developed. The structuring techniques available so far (like modules, classes, parametric polymorphism) are still not enough or not appropriate for some application areas. In symbolic computation, in particular computer algebra, several problems occur that are dicult to handle with any existing programming language. Indeed, nearly all available computer algebra systems suer from the fact that the underlying programming language imposes too many restrictions. We propose to develop a language that combines the essential features from functional languages, object-oriented languages, and computer algebra systems in a semantically clean manner. Although intended for use in symbolic computation, this language should prove interesting as a general purpose programming language. The main innovation will be the application of sophisticated type systems to the needs of computer algebra systems. We will demonstrate the capabilities of the language by using it to implement a small computer algebra library. This implementation will be compared against a straightforward Lisp implementation and against existing computer algebra systems. Our development should have an impact both on the programming languages world and on the computer algebra world.

[Beno86] Michael Ben-Or, Dexter Kozen, and John Reif. The complexity of elementary algebra and geometry. *J. Computer and System Sciences*, 32(2):251–264, 1986.

**Abstract:** The theory of real closed fields can be decided in exponential space or parallel exponential time. In fixed dimesnion, the theory can be decided in NC.

[Berg92] Emery Berger. FP + OOP = Haskell. Technical Report TR-92-30, University of Texas, 1992.

**Abstract:** The programming language Haskell adds object-oriented functionality (using a concept known as type classes) to a pure func-

tional programming framework. This paper describes these extensions and analyzes its accomplishments as well as some problems.

[Bern91] Paul Bernays. *Axiomatic Set Theory*. Dover, 1991.

[Bert95] Laurent Bertrand. Computing a hyperelliptic integral using arithmetic in the jacobian of the curve. *Applicable Algebra in Engineering, Communication and Computing*, 6:275–298, 1995.

**Abstract:** In this paper, we describe an efficient algorithm for computing an elementary antiderivative of an algebraic function defined on a hyperelliptic curve. Our algorithm combines B.M. Trager's integration algorithm and a technique for computing in the Jacobian of a hyperelliptic curve introduced by D.G. Cantor. Our method has been implemented and successfully compared to Trager's general algorithm.

[Beth91] T. Beth, W. Geiselmann, and F. Meyer. Finding (Good) Normal Bases in Finite Fields. *Proc. ISSAC '91*, 1991.

[Bhat19] Siddharth Bhat. Computing Equivalent Gate Sets Using Grobner Bases, 2019.

**Link:** https://bollu.github.io/#computing-equivalent-gat-sets-using-grobner-bases

[Birt80] Graham M. Birtwistle. *Simula Begin*. Chartwell-Bratt, 1980, 9780862380090.

[Brea89] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance and Explicit Coercion. In *Logic in Computer Science*, 1989, 0-8186-1954-6.

**Abstract:** A method is presented for providing semantic interpretations for languages which feature inheritance in the framework of statically checked, rich type disciplines. The approach is illustrated by an extension of the language Fun of L. Cardelli and P. Wegner (1985), which is interpreted via a translation into an extended polymorphic lambda calculus. The approach interprets inheritances in Fun as coercion functions already definable in the target of the translation. Existing techniques in the theory of semantic domains can then be used to interpret the extended polymorphic lambda calculus, thus providing many models for the original language. The method allows the simultaneous modeling of parametric polymorphism, recursive types, and inheritance, which has been regarded as problematic because of the seemingly contradictory characteristics of inheritance and type recursion on higher types. The main difficulty in providing interpretations for explicit type disciplines featuring inheritance is identified. Since interpretations follow the type-checking derivations, coherence theorems are required, and the authors prove them for their semantic method.

[Brea89a] Jean Breazu-Tannen, Val Gallier. Polymorphic Rewriting Concerves Algebraic Strong Normalization and Confluence. In *Automata, Languages and Programming*, pages 137–150, 1989.

**Abstract:** We study combinations of many-sorted algebraic term rewriting systems and polymorphic lambda term rewriting. Algebraic and lambda terms are mixed by adding the symbols of the algebraic signature to the polymorphic lambda calculus, as higher-

order constants. We show that if a many-sorted algebraic rewrite system R is strongly normalizing (terminating, noetherian), then $R+\beta+\nu+type-\beta+type-\nu$ rewriting of mixed terms is also strongly normalizing. We obtain this results using a technique which generalizes Girard's "candidats de reductibilitié", introduced in the original proof of strong normalization for the polymorphic lambda calculus. We also show that if a many-sorted algebraic rewrite system $R$ has the Church-Rosser property (is confluent), then $R+\beta+type-\beta+type-\nu$ rewriting of mixed terms has the Church-Rosser property too. Combining the two results, we conclude that if $R$ is canonical (complete) on algebraic terms, then $R+\beta+type-\beta+type-\nu$ is canonical on mixed terms. $\nu$ reduction does not commute with algebraic reduction, in general. However, using long $\nu$-normal forms, we show that if $R$ is canonical then $R+\beta+type-\beta+type-\nu$ convertibility is still decidable.

[Brea91] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as Implicit Coercion. *Information and Computation*, 93(1):172–221, 1991.

**Abstract:** We present a method for providing semantic interpretations for languages with a type system featuring inheritance polymorphism. Our approach is illustrated on an extension of the language Fun of Cardelli and Wegner, which we interpret via a translation into an extended polymorphic lambda calculus. Our goal is to interpret inheritances in Fun via coercion functions which are definable in the target of the translation. Existing techniques in the theory of semantic domains can be then used to interpret the extended polymorphic lambda calculus, thus providing many models for the original language. This technique makes it possible to model a rich type discipline which includes parametric polymorphism and recursive types as well as inheritance. A central difficulty in providing interpretations for explicit type disciplines featuring inheritance in the sense discussed in this paper arises from the fact that programs can type-check in more than one way. Since interpretations follow the type-checking derivations, coherence theorems are required: that is, one must prove that the meaning of a program does not depend on the way it was type-checked. Proofs of such theorems for our proposed interpretation are the basic technical results of this paper. Interestingly, proving coherence in the presence of recursive types, variants, and abstract types forced us to reexamine fundamental equational properties that arise in proof theory (in the form of commutative reductions) and domain theory (in the form of strict vs. non-strict functions).

[Bro98b] Manuel Bronstein. Symbolic Integration Tutorial, 1998.

**Link:** http://www-sop.inria.fr/cafe/Manuel.Bronstein/publications/issac98.pdf

[Bron90c] Manuel Bronstein. On the integration of elementary functions. *Journal of Symbolic Computation*, 9(2):117–173, February 1990.

[Bron91a] M. Bronstein. The Risch Differential Equation on an Algebraic Curve. In *Proc. 1991 Int. Symp. on Symbolic and Algebraic Computation*, ISSAC'91, pages 241–

246. ACM, NY, 1991, 0-89791-437-6.

**Abstract:** We present a new rational algorithm for solving Risch differential equations over algebraic curves. This algorithm can also be used to solve $n^{th}$-order linear ordinary differential equations with coefficients in an algebraic extension of the rational functions. In the general ("mixed function") case, this algorithm finds the denominator of any solution of the equation.

[Bron97] Manuel Bronstein. *Symbolic Integration I–Transcendental Functions*. Springer, Heidelberg, 1997, 3-540-21493-3.

**Link:** http://evil-wire.org/arrrXiv/Mathematics/Bronstein,_Symbolic_Integration_I,1997.

[Bron98] Manuel Bronstein. The lazy hermite reduction. Rapport de Recherche RR-3562, French Institute for Research in Computer Science, 1998.

**Abstract:** The Hermite reduction is a symbolic integration technique that reduces algebraic functions to integrands having only simple affine poles. While it is very effective in the case of simple radical extensions, its use in more general algebraic extensions requires the precomputation of an integral basis, which makes the reduction impractical for either multiple algebraic extensions or complicated ground fields. In this paper, we show that the Hermite reduction can be performed without *a priori* computation of either a primitive element or integral basis, computing the smallest order necessary for a particular integrand along the way.

[Brow01a] Christopher W. Brown. Simple CAD Construction and its Applications. *J. Symbolic Computation*, 31:521–547, 2001.

**Abstract:** This paper presents a method for the simplification of truth-invariant cylindrical algebraic decompositions (CADs). Examples are given that demonstrate the usefulness of the method in speeding up the solutoin formula construction phase of the CAD-based quantifier elimination algorithm. Applications of the method to the construction of truth-invariant CADs for very large quantifier-free formulas and quantifier elimination of non-prenex formulas are also discussed.

[Brow99] Christopher W. Brown. *Solution Formula Construction for Truth Invariant CADs*. PhD thesis, University of Delaware, 1999.

**Abstract:** The CAD-based quantifier elimination algorithm takes a formula from the elementary theory of real closed fields as input, and constructs a CAD of the space of the formula's unquantified variables. This decomposition is truth invariant with respect to the input formula, meaning that the formula is either identically true or identically false in each cell of the decomposition. The method determines the truth of the input formula for each cell of the CAD, and then uses the CAD to construct a solution formula – a quantifier free formula that is equivalent to the input formula. This final phase of the algorithm, the solution formula construction phase, is the focus of this thesis. An optimal solution formula construction algorithm would be *complete* – i.e. applicable to any truth-invariant CAD, would be *efficient*,

and would produce *simple* solution formulas. Prior to this thesis, no method was available with even two of these three properties. Several algorithms are presented, all addressing problems related to solution formula construction. In combination, these provide an efficient and complete method for constructing solution formulas that are simple in a variety of ways. Algorithms presented in this thesis have been implemented using the SACLIB library, and integrated into QEPCAD, a SACLIB-based implementation of quantifier elimination by CAD. Example computations based on these implementations are discussed.

**Link:** http://www.usna.edu/Users/cs/wcbrown/research/thesis.ps.gz

[Broy88] Manfried Broy. Equational Specification of Partial Higher-order Algebras. *Theoretical Computer Science*, 57(1):3–45, 1988.

**Abstract:** The theory of algebraic abstract types specified by positive conditional formulas formed of equations and a definedness predicate is outlined and extended to hierarchical types with "noustrict" operations, partial and even infinite objects. Its model theory is based on the concept of partial interpretations. Deduction rules are given, too. Models of types are studied where all explicit equations have solutions. The inclusion of nigher-order types, i.e., types comprising higher-order functions leads to an algebraic ("equational") specification of algebras including sorts with "infinite" objects and higher-order functions ("functionals").

[Bruc92] Kim Bruce and John C. Mitchell. PER Models of Subtyping, Recursive Types and Higher-Order Polymorphism. In *POPL '92*, pages 316–327, 1992.

**Abstract:** We relate standard techniques for solving recursive domain equations to previous models with types interpreted as partial equivalence relations (per's) over a $D_\infty$ lambda model. This motivates a particular choice of type functions, which leads to an extension of such models to higher-order polymorphism. The resulting models provide natural interpretations for function spaces, records, recursively defined types, higher-order type functions, and bounded polymorphic types $\forall X <: Y.A$ where the bound may be of a higher kind. In particular, we may combine recursion and polymorphism in a way that allows the bound $Y$ in $\forall X <: Y.A$ to be recursively defined. The model may also be used to interpret so-called "F-bounded polymorphism". Together, these features allow us to represent several forms of type and type functions that seem to arise naturally in typed object-oriented programming.

[Bruc93] Kim B. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *POPL 93*, pages 285–298, 1993, 0-89791-560-7.

**Abstract:** In this paper we introduce a statically-typed, functional, object-oriented programming language, TOOPL, which supports classes, objects, methods, instance variable, subtypes, and inheritance. It has proved to be surprisingly difficult to design statically-typed object-oriented languages which are nearly as expressive as Smalltalk and yet have no holes in their typing systems. A particular problem with statically type checking object-oriented languages

is determining whether a method provided in a superclass will continue to type check when inherited in a subclass. This program is solved in our language by providing type checking rules which guarantee that a method which type checks as part of a class will type check correctly in all legal subclasses in which it is inherited. This feature enables library providers to provide only the interfaces of classes with executables and still allow users to safely create subclasses. The design of TOOPL has been guided by an analysis of the semantics of the language, which is given in terms of a sufficiently rich model of the F-bounded second-order lambda calculus. This semantics supported the language design by providing a means of proving that the type-checking rules for the language are sound, ensuring that well-typed terms produce objects of the appropriate type. In particular, in a well-typed program it is impossible to send a message to an object which lacks a corresponding method.

[Buch82] Bruno Buchberger, George Edwin Collins, and Rudiger Loos. *Computer Algebra: Symbolic and Algebraic Computation*. Springer, 1982, 978-3-211-81684-4.

[Buch93] Bruno Buchberger, George E. Collins, Mark J. Encarnacion, Hoon Hong, Jeremy R. Johnson, Werner Krandick, Rudiger Loos, Ana M. Mandache, Andreas Neubacher, and Herbert Vielhaber. SACLIB 1.1 User's Guide. Technical report, Kurt Godel Institute, 1993.

**Abstract:** This paper lists most of the algorithms provided by SACLIB and shows how to call them from C. There is also a brief explanation of the inner workings of the list processing and garbage collection facilities of SACLIB

[Buen91] R. Buendgen, G. Hagel, R. Loos, S. Seitz, G. Simon, R. Stuebner, and A. Weber. SAC-2 in ALDES – Ein Werkzeug fur dis Algorithmenforschung. *MathPAD 1*, 3:33–37, 1991.

[Bund93] Reinhard Bundgen. *The ReDuX System Documentation*. WSI, 1993.

[Bund93a] Reinhard Bundgen. Reduce the Redex $->$ ReDuX. In *Proc. Rewriting Techniques and Applications 93*, pages 446–450. Springer-Verlag, 1993, 3-540-56868-9.

[Butl90] Greg Butler and John Cannon. The Design of Cayley – A Language for Modern Algebra. In *DISCO 1990*, pages 10–19, 1990.

**Abstract:** Established practice in the domain of modern algebra has shaped the design of Cayley. The design has also been responsive to the needs of its users. The requirements of the users include consistency with common mathematical notation; appropriate data types such as sets, sequences, mappings, algebraic structures and elements; efficiency; extensibility; power of in-built functions and procedures for known algorithms; and access to common examples of algebraic structures. We discuss these influences on the design of Cayley's user language.

[Cann01] John J. Cannon and Catherine Playoust. *An Introduction to Algebraic Programming with Magma*. University of Sydney, 2001.

[Cann87] John Canny. A new algebraic method of robot motion planning and real geom-

etry. In *IEEE Symp. on Foundations of Comp. Sci.*, pages 39–48, 1987.

**Abstract:** We present an algorithm which solves the findpath or generalized movers' problem in single exponential sequential time. This is the first algorithm for the problem whose sequential time bound is less than double exponential. In fact, the combinatorial exponent of the algorithm is equal to the number of degrees of freedom, making it worst-case optimal, and equaling or improving the time bounds of many special purpose algorithms. The algorithm accepts a formula for a semi-algebraic set S describing the set of free configurations and produces a one-dimensional skeleton or "roadmap" of the set, which is connected within each connected component of S. Additional points may be linked to the roadmap in linear time. Our method draws from results of singularity theory, and in particular makes use of the notion of stratified sets as an efficient alternative to cell decomposition. We introduce an algebraic tool called the multivariate resultant which gives a necessary and sufficient condition for a system of homogeneous polynomials to have a solution, and show that it can be computed in polynomial parallel time. Among the consequences of this result are new methods for quantifier elimination and an improved gap theorem for the absolute value of roots of a system of polynomials.

[Cann88] John Canny. Some algebraic and geometric computations in PSPACE. In *Proc 20th ACM Symp. on the theory of computing*, pages 460–467, 1988, 0-89791-264-0.

**Abstract:** We give a PSPACE algorithm for determining the signs of multivariate polynomials at the common zeros of a system of polynomial equations. One of the consequences of this result is that the "Generalized Movers' Problem" in robotics drops from EXPTIME into PSPACE, and is therefore PSPACE-complete by a previous hardness result [Rei]. We also show that the existential theory of the real numbers can be decided in PSPACE. Other geometric problems that also drop into PSPACE include the 3-d Euclidean Shortest Path Problem, and the "2-d Asteroid Avoidance Problem" described in [RS]. Our method combines the theorem of the primitive element from classical algebra with a symbolic polynomial evaluation lemma from [BKR]. A decision problem involving several algebraic numbers is reduced to a problem involving a single algebraic number or primitive element, which rationally generates all the given algebraic numbers.

**Link:** http://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-88-439.pdf

[Cann89] J. Canny, E. Kaltofen, and Lakshman Yagati. Solving systems of non-linear polynomial equations faster. In *Proc. 1989 Internat. Symp. Symbolic Algebraic Comput.*, pages 121–128, 1989.

**Link:** http://www.math.ncsu.edu/~kaltofen/bibliography/89/CKL89.pdf

[Cann93] John Canny. Improved algorithms for sign and existential quantifier elimination. *The Computer Journal*, 36:409–418, 1993.

**Abstract:** Recently there has been a lot of activity in algorithms that work over real closed fields, and that perform such calculations as quantifier elimination or computing connected components of semi-

algebraic sets. A cornerstone of this work is a symbolic sign determination algorithm due to Ben-Or, Kozen and Reif. In this paper we describe a new sign determination method based on the earlier algorithm, but with two advantages: (i) It is faster in the univariate case, and (ii) In the general case, it allows purely symbolic quantifier elimination in pseudo-polynomial time. By purely symbolic, we mean that it is possible to eliminate a quantified variable from a system of polynomials no matter what the coefficient values are. The previous methods required the coefficients to be themselves polynomials in other variables. Our new method allows transcendental functions or derivatives to appear in the coefficients. Another corollary of the new sign-determination algorithm is a very simple, practical algorithm for deciding existentially-quantified formulae of the theory of the reals. We present an algorithm that has a bit complexity of $n^{k+1}d^{O(k)}(clogn)^{1+\epsilon}$ randomized, or $n^{n+1}d^{O(k^2)}c(1+\epsilon)$ deterministic, for any $\epsilon > 0$, where $n$ is the number of polynomial constraints in the defining formula, $k$ is the number of variables, $d$ is a bound on the degree, $c$ bounds the bit length of the coefficient. The algorithm makes no general position assumptions, and its constants are much smaller than other recent quantifier elimination methods.

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

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

illustrative examples. We christen this language Fun because fun instead of $\lambda$ is the functional abstraction keyword and because it is pleasant to deal with. Fun is mathematically simple and can serve as a basis for the design and implementation of real programming languages with type facilities that are more powerful and expressive than those of existing programming languages. In particular, it provides a basis for the design of strongly typed object-oriented languages

[Card86] Luca Cardelli. Typechecking Dependent Types and Subtypes. In *Foundations of Logic and Functional Programming*, volume 523, pages 45–57, 1996.

**Link:** http://lucacardelli.name/Papers/Dependent%20Typechecking.US.pdf

[Card88] Luca Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76(2-3):138–164, 1988.

[Card88a] Luca Cardelli. Structural Subtyping and the Notion of Power Type. In *POPL '88*. ACM, 1988.

[Card91] Luca Cardelli and Giuseppe Longo. A Semantic Basis for Quest. *J. of Functional Programming*, 1(4):417–458, 1991.

**Abstract:** Quest is a programming language based on impredicative type quantifiers and subtyping within a three-level structure of kinds, types and type operators, and values. The semantics of Quest is rather challenging. In particular, difficulties arise when we try to model simultaneously features such as contravariant function spaces, record types, subtyping, recursive types and fixpoints. In this paper we describe in detail the type inference rules for Quest, and give them meaning using a partial equivalence relation model of types. Subtyping is interpreted as in previous work by Bruce and Longo (1989), but the interpretation of some aspects namely subsumption, power kinds, and record subtyping is novel. The latter is based on a new encoding of record types. We concentrate on modelling quantifiers and subtyping; recursion is the subject of current work.

[Cavi98] B. F. Caviness and J. R. Johnson. *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Springer, 1998, 3-221-82794-3.

[Chan90] C.C. Chang and H. Jerome Keisler. *Model Theory*, volume 73. North Holland, 1990.

**Abstract:** Since the second edition of this book (1977), Model Theory has changed radically, and is now concerned with fields such as classification (or stability) theory, nonstandard analysis, model-theoretic algebra, recursive model theory, abstract model theory, and model theories for a host of nonfirst order logics. Model theoretic methods have also had a major impact on set theory, recursion theory, and proof theory. This new edition has been updated to take account of these changes, while preserving its usefulness as a first textbook in model theory. Whole new sections have been added, as well as new exercises and references. A number of updates, improvements and corrections have been made to the main text

**Comment:** Studics in Logic and the Foundations of Mathematics

[Char91] Bruce Char, Keith O. Geddes, Gaston H. Gonnet, Benton Leong, Michael B.

Monagan, and Stephen M. Watt. *Maple V Language Reference Manual*. Springer, 1991, 978-0-387-94124-0.

[Char91a] Bruce Char, Keith O. Geddes, Gaston H. Gonnet, Benton Leong, Michael B. Monagan, and Stephen M. Watt. *Maple V Library Reference Manual*. Springer, 1991, 978-1-4757-2133-1.

**Abstract:** The design and implementation of the Maple system is an on-going project of the Symbolic Com putation Group at the University of Waterloo in Ontario, Canada. This manual corresponds with version V (roman numeral five) of the Maple system. The online help subsystem can be invoked from within a Maple session to view documentation on specific topics. In particular, the command ?updates points the user to documentation updates for each new version of Maple. The Maple project was first conceived in the autumn of 1980, growing out of discussions on the state of symbolic computation at the University of Waterloo. The authors wish to acknowledge many fruitful discussions with colleagues at the University of Waterloo, particularly Morven Gen tleman, Michael Malcolm, and Frank Tompa. It was recognized in these discussions that none ofthe locaIly-available systems for symbolic computation provided the facilities that should be expected for symbolic computation in modern computing environments. We concluded that since the basic design decisions for the then-current symbolic systems such as ALTRAN, CAMAL, REDUCE, and MACSYMA were based on 1960's computing technology, it would be wise to design a new system "from scratch//. Thus we could take advantage of the software engineering technology which had become available in recent years, as well as drawing from the lessons of experience. Maple's basic features (elementary data structures, Input/output, arithmetic with numbers, and elementary simplification) are coded in a systems programming language for efficiency.

[Chen92] Kung Chen, Paul Hudak, and Martin Odersky. Parametric Type Classes. In *Proc. ACM Conf. on LISP and Functional Programming*, pages 170–181, 1992.

**Abstract:** We propose a generalization to Haskell's type classes where a class can have type parameters besides the placeholder variable. We show that this generalization is essential to represent container classes with overloaded data constructor and selector operations. We also show that the resulting type system has principal types and present unification and type reconstruction algorithms.

[Coll02] George E. Collins, Jeremy R. Johnson, and Werner Krandick. Interval arithmetic in cylindrical algebraic decomposition. *J. Symbolic Computation*, 34(2):145–157, 2002.

**Abstract:** Cylindrical algebraic decomposition requires many very time consuming operations, including resultant computation, polynomial factorization, algebraic polynomial gcd computation and polynomial real root isolation. We show how the time for algebraic polynomial real root isolation can be greatly reduced by using interval arithmetic instead of exact computation. This substantially reduces

the overall time for cylindrical algebraic decomposition.

[Coll75] George E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. *Lecture Notes in Computer Science*, 33:134–183, 1975.

**Abstract:** I. Introduction. Tarski in 1948, published a quantifier elimination method for the elementary theory of real closed fields (which he had discoverd in 1930). As noted by Tarski, any quantifier elimination method for this theory also provides a decision method, which enables one to decide whether any sentence of the theory is true or false. Since many important and difficult mathematical problems can be expressed in this theory, any computationally feasible quantifier elimination algorithm would be of utmost significance. However, it became apparent that Tarski's method required too much computation to be practical except for quite trivial problems. Seidenberg in 1954, described another method which he thought would be more efficient. A third method was published by Cohen in 1969. Some significant improvements of Tarski's method have been made by W. Boge, which are described in a thesis by Holthusen This paper describes a completely new method which I discoverd in February 1973. This method was presented in a seminar at Stanford University in March 1973 and in abstract form at a symposium at Carnegie-Mellon University in May 1973. In August 1974 a full presentation of the method was delivered at the EUROSAM 74 Conference in Stockholm, and a preliminary version of the present paper was published in the proceedings of that conference.

[Coll90] George E. Collins and Rudiger Loos. Specification and Index of SAC-2 Algorithms. technical report WSI-90-4, Univ. of Tubingen, 1990.

[Coll91] George E. Collins and Hoon Hong. Partial Cylindrical Algebraic Decomposition for Quantifier Elimination. *J. Symbolic Computation*, 12:299–328, 1991.

**Abstract:** The Cylindrical Algebraic Decomposition method (CAD) decomposes $R^r$ into regions over which given polynomials have constant signs. An important application of GAD is quantifier elimination in elementary algebra and geometry. In this paper we present a method which intermingles CAD construction with truth evaluation so that parts of the CAD are constructed only as needed to further truth evaluation and aborts CAD construction as soon as no more truth evaluation is needed. The truth evaluation utilizes in an essential way any quantifiers which are present and additionally takes account of atomic formulas from which some variables are absent. Preliminary observations show that the new method is always more efficient than the original, and often significantly more efficient.

[Coll98] George E. Collins. Quantifier Elimination by Cylindrical Algebraic Decomposition – Twenty Years of Progress. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 8–23, 1998, 3-211-82794-3.

**Abstract:** The CAD (cylindrical algebraic decomposition) method and its application to QE (quantifier elimination) for ERA (elementary real algebra) was announced by the author in 1973 at Carnegie

Mellon University (Collins 1973b). In the twenty years since then several very important improvements have been made to the method which, together with a very large increase in available computational power, have made it possible to solve in seconds or minutes some interesting problems. In the following we survey these improvements and present some of these problems with their solutions.

[Como90] Hubert Comon. Equational Formulas in Order-sorted Algebras. In *IICALP 90. Automata, Languages and Programming*, pages 674–688, 1990.

**Abstract:** We propose a set of transformation rules for first order formulas whose atoms are either equations between terms or sort constraints t s where s is a regular tree language (or a sort in the algebraic specification community). This set of rules is proved to be correct, terminating and complete. This shows in particular that the first order theory of any rational tree language is decidable, extending the results of [Mal71,CL89,Mah88]. We also show how to apply our results to automatic inductive proofs in equational theories.

[Como91] Hubert Comon, D. Lugiez, and Ph. Schnoebelen. A Rewrite-based Type Discipline for a Subset of Computer Algebra. *J. Symbolic Computation*, 11(4):349–368, 1991.

**Abstract:** This paper is concerned with the type structure of a system including polymorphism, type properties and subtypes. This type system originates from computer algebra but it is not intended to be the solution of all type problems in this area. Types (or sets of types) are denoted by terms in some order-sorted algebra. We consider a rewrite relation in this algebra, which is intended to express subtyping. The relations between the semantics and the axiomatization are investigated. It is shown that the problem of type inference is undecidable but a narrowing strategy for semi-decision procedures is described and studied.

[Cons85] R.L. Constable, S.F. Allen, H.M. Bromley, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panagaden, J.T. Tsaaki, and S.F. Smith. *Implementing Mathematics with The Nuprl Proof Development System.* Prentice-Hall, 1985.

[Cool92] Kris Coolsaet. A Quick Introduction to the Programming Language MIKE. *Sigplan Notices*, 27(6):37–48, 1992.

**Abstract:** MIKE is a new programming language developed by the author as a base language for the development of algebraic and symbolic algorithms. It is a structured programming language with a MODULA-2-like syntax supporting special features such as transparent dynamic memory management, discriminated union types, operator overloading, data abstraction and parametrized types. This text gives an overview of the main features of the language as of version 2.0.

[Coqu86] Thierry Coquand. An Analysis of Girard's Paradox. Technical Report 531, INRIA Centre de Rocquencourt, 1986.

**Abstract:** We study the consistency of a few formal systems specially

some extensions of Church's calculus and the construction system. We show that Church's calculus is not compatible with the notion of second-order type. We apply this result for showing that the calculus of construction wit four levels is inconsistent. We suggest finally some consistent extensions of these two calculi.

[Crol93] R.L. Crole. *Categories for Types*. Cambridge University Press, 1993, 978-0521457019.

[Dalm92] Stephane Dalmas. A polymorphic functional language applied to symbolic computation. In *Proc. ISSAC 1992*, ISSAC 1992, pages 369–375, 1992, 0-89791-489-9 (soft cover) 0-89791-490-2 (hard cover).

**Abstract:** The programming language in which to describe mathematical objects and algorithms is a fundamental issue in the design of a symbolic computation system. XFun is a strongly typed functional programming language. Although it was not designed as a specialized language, its sophisticated type system can be successfully applied to describe mathematical objects and structures. After illustrating its main features, the author sketches how it could be applied to symbolic computation. A comparison with Scratchpad II is attempted. XFun seems to exhibit more flexibility simplicity and uniformity.

[Dama82] Luis Damas and Robin Milner. Principal Type-schemes for Functional Programs. In *Proc POPL '82*, pages 207–212, 1982.

[Damg91] I. Damgard and P. Landrock. Improved Bounds for the Rabin Primality Test, 1991.

[Dave87] James H. Davenport and G.C. Smith. Rabin's Primality Testing Algorithm – a Group Theory View, 1987.

[Dave88] James H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra: Systems and Algorithms for Algebraic Computation*. Academic Press, 1988, 0-12-204230-1.

**Link:** http://staff.bath.ac.uk/masjhd/masternew.pdf

[Dave90] James H. Davenport and Barry M. Trager. Scratchpad's view of algebra I: Basic commutative algebra. In *Design and Implementation of Symbolic Computation Systems*, DISCO '90, pages 40–54. Springer-Verlag, 1990, 0-387-52531-9.

**Abstract:** While computer algebra systems have dealt with polynomials and rational functions with integer coefficients for many years, dealing with more general constructs from commutative algebra is a more recent problem. In this paper we explain how one system solves this problem, what types and operators it is necessary to introduce and, in short, how one can construct a computational theory of commutative algebra. Of necessity, such a theory is rather different from the conventional, non-constructive, theory. It is also somewhat different from the theories of Seidenberg [1974] and his school, who are not particularly concerned with practical questions of efficiency.

**Comment:** AXIOM Technical Report, ATR/1, NAG Ltd., Oxford, 1992

**Link:** http://opus.bath.ac.uk/32336/1/Davenport_DISCO_1990.pdf

[Dave91] J. H. Davenport, P. Gianni, and B. M. Trager. Scratchpad's View of Algebra II:

A Categorical View of Factorization. In *Proc. 1991 Int. Symp. on Symbolic and Algebraic Computation*, ISSAC '91, pages 32–38, New York, NY, USA, 1991, 0-89791-437-6. ACM.

**Abstract:** This paper explains how Scratchpad solves the problem of presenting a categorical view of factorization in unique factorization domains, i.e. a view which can be propagated by functors such as SparseUnivariatePolynomial or Fraction. This is not easy, as the constructive version of the classical concept of UniqueFactorizationDomain cannot be so propagated. The solution adopted is based largely on Seidenberg's conditions (F) and (P), but there are several additional points that have to be borne in mind to produce reasonably efficient algorithms in the required generality. The consequence of the algorithms and interfaces presented in this paper is that Scratchpad can factorize in any extension of the integers or finite fields by any combination of polynomial, fraction and algebraic extensions: a capability far more general than any other computer algebra system possesses. The solution is not perfect: for example we cannot use these general constructions to factorize polyinmoals in $\overline{Z[\sqrt{-5}]}[x]$ since the domain $\overline{Z[\sqrt{-5}]}$ is not a unique factorization domain, even though $\overline{Z[\sqrt{-5}]}$ is, since it is a field. Of course, we can factor polynomials in $\overline{Z}[\sqrt{-5}][x]$

**Link:** http://doi.acm.org/10.1145/120694.120699

[Dave92] James H. Davenport. Primality Testing Revisited. In *Proc. ISSAC 1992*, ISSAC 92, pages 123–129. ACM, 1992.

**Abstract:** Rabin's algorithm is commonly used in computer algebra systems and elsewhere for primality testing. This paper presents an experience with this in the Axiom computer algebra system. As a result of this experience, we suggest certain strengthenings of the algorithm.

**Link:** http://staff.bath.ac.uk/masjhd/ISSACs/ISSAC1992.pdf

**Algebra:**
(p**??**) package PRIMES IntegerPrimesPackage

[Dave93a] James H. Davenport. C9: Universal Algebra, 1993.

**Comment:** Lecture Notes for 2nd year undergrad and mather's course in Universal Algebra

[Davi94] Martin D. Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Academic Press, 1994, 978-0122063824.

[Ders89] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite Systems. Technical Report 478, Laboratoire de Recherche en Informatique, 1989.

**Link:** http://www.cs.tau.ac.il/~nachum/papers/survey-draft.pdf

[Devl79] Keith J. Devlin. *Fundamentals of Contemporary Set Theory*. Springer-Verlag, 1979, 978-0387904412.

[Dolz04] Andreas Dolzmann, Andreas Seidl, and Thomas Sturm. Efficient projection

orders for CAD. In *proc ISSAC'04*, pages 111–118. ACM, 2004, 1-58113-827-X.

**Abstract:** We introduce an efficient algorithm for determining a suitable projection order for performing cylindrical algebraic decomposition. Our algorithm is motivated by a statistical analysis of comprehensive test set computations. This analysis introduces several measures on both the projection sets and the entire computation, which turn out to be highly correlated. The statistical data also shows that the orders generated by our algorithm are significantly close to optimal.

[Doye97] Nicolas James Doye. *Order Sorted Computer Algebra and Coercions.* PhD thesis, University of Bath, 1997.

**Abstract:** Computer algebra systems are large collections of routines for solving mathematical problems algorithmically, efficiently and above all, symbolically. The more advanced and rigorous computer algebra systems (for example, Axiom) use the concept of strong types based on order-sorted algebra and category theory to ensure that operations are only applied to expressions when they "make sense". In cases where Axiom uses notions which are not covered by current mathematics we shall present new mathematics which will allow us to prove that all such cases are reducible to cases covered by the current theory. On the other hand, we shall also point out all the cases where Axiom deviates undesirably from the mathematical ideal. Furthermore we shall propose solutions to these deviations. Strongly typed systems (especially of mathematics) become unusable unless the system can change the type in a way a user expects. We wish any change expected by a user to be automated, "natural", and unique. "Coercions" are normally viewed as "natural type changing maps". This thesis shall rigorously define the word "coercion" in the context of computer algebra systems. We shall list some assumptions so that we may prove new results so that all coercions are unique. This concept is called "coherence". We shall give an algorithm for automatically creating all coercions in type system which adheres to a set of assumptions. We shall prove that this is an algorithm and that it always returns a coercion when one exists. Finally, we present a demonstration implementation of this automated coerion algorithm in Axiom.

[Duva95] Dominique Duval. Dynamic evaluation and algebraic closure in Axiom. *Journal of Pure and Applied Algebra*, 99:267–295, 1995.

**Abstract:** Dynamic evaluation allows to compute with algebraic numbers without factorizing polynomials. It also allows to manipulate parameters in a flexible and user-friendly way. The aim of this paper is the following: Explain what is dynamic evaluation, with its basic notions of dynamic set and splitting. Present its application to computations involving algebraic numbers, which amounts to defining the dynamic algebraic closure of a field. Describe the Axiom program which implements this, and give a user guide for it (only this last point assumes some knowledge of Axiom) Dynamic evaluation is described here without any reference to sketch theory, however our presentation,

less rigourous, may be considered as more accessible.

**Comment:** Evaluation dynamique et clôture algébrique en Axiom

[Ehri85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer Verlag, 1985, 0-387-13718-1.

[Elli89] Conal M. Elliott. Higher-order Unification with Dependent Function Types. In *Rewriting Techniques and Applications*, pages 121–136, 1989.

**Abstract:** Roughly fifteen years ago, Huet developed a complete semidecision algorithm for unification in the simply typed $\lambda$-calculus ($\lambda_{\rightarrow}$). In spite of the undecidability of this problem, his algorithm is quite usable in practice. Since then, many important applications have come about in such areas as theorem proving, type inference, program transformation, and machine learning. Another development is the discovery that by enriching $\lambda_{\rightarrow}$ to include *dependent function types*, the resulting calculus ($\lambda_{\Pi}$) forms the basis of a very elegant and expressive Logical Framework, encompassing the syntax, rules, and proofs for a wide class of logics. This paper presents an algorithm in the spirit of Huet's, for unification in $\lambda_{\Pi}$. This algorithm gives us the best of both worlds: the automation previously possible in $\lambda_{\rightarrow}$ and the greatly enriched expressive power of $\lambda_{\Pi}$. It can be used to considerable advantage in many of the current applications of Huet's algorithm, and has important new applications as well. These include automated and semi-automated theorem proving in encoded logics, and automatic type inference in a variety of encoded languages.

[Farm90] William M. Farmer. A Partial Functions Version of Church's Simple Theory of Types. *The Journal of Symbolic Logic*, 55(3):1269–1291, 1990.

**Abstract:** Church's simple theory of types is a system of higher-order logic in which functions are assumed to be total. We present in this paper a version of Church's system called PF in which functions may be partial. The semantics of PF, which is based on Henkin's general-models semantics, allows terms to be nondenoting but requires formulas to always denote a standard truth value. We prove that PF is complete with respect to its semantics. The reasoning mechanism in PF for partial functions corresponds closely to mathematical practice, and the formulation of PF adheres tightly to the framework of Church's system.

[Fate90] Richard J. Fateman. Advances and trends in the design and construction of algebraic manipulation systems. In *Proc. ISSAC 1990*, pages 60–67. ACM, 1990, 0-89791-401-5.

**Abstract:** We compare and contrast several techniques for the implementation of components of an algebraic manipulation system. On one hand is the mathematical-algebraic approach which chaaracterizes (for example) IBM's Axiom. On the other hand is the more *ad hoc* approach which characterizes many other popular systems (for example, Macsyma, Reduce, Maple, and Mathematica). While the algebraic approach has generally positive results, careful examination suggests that there are significant remaining problems, expecially in the representation and manipulation of analytical, as opposed to al-

gebraic, mathematics. We describe some of these problems and some general approaches for solutions.

**Link:** http://people.eecs.berkeley.edu/~fateman/papers/advances.pdf

[Faxe02] Karl-Filip Faxen. A Static Sematics for Haskell. *J. Functional Programming*, 12(4-5):295–357, 2002.

**Abstract:** This paper gives a static semantics for Haskell 98, a non-strict purely functional programming language. The semantics formally speci es nearly all the details of the Haskell 98 type system, including the resolution of overloading, kind inference (including defaulting) and polymorphic recursion, the only major omission being a proper treatment of ambiguous overloading and its resolution. Overloading is translated into explicit dictionary passing, as in all current implementations of Haskell. The target language of this translation is a variant of the Girard-Reynolds polymorphic lambda calculus featuring higher order polymorphism and explicit type abstraction and application in the term language. Translated programs can thus still be type checked, although the implicit version of this system is impredicative. A surprising result of this formalization e ort is that the monomorphism restriction, when rendered in a system of inference rules, compromises the principal type property.

[Fitc87] N. Fitchas, A. Galligo, and J. Morgenstern. Algorithmes repides en séquential et en parallele pour l'élimination de quantificateurs en géométrie élémentaire. technical report, UER de Mathématiques Universite de Paris VII, 1987.

[Flet01] John P. Fletcher. Symbolic processing of Clifford Numbers in C++, 2001.

[Flet09] John P. Fletcher. Clifford Numbers and their inverses calculated using the matrix representation.

**Link:** http://www.ceac.aston.ac.uk/research/staff/jpf/papers/paper24/index.php

[Fode83] John K. Foderaro. *The Design of a Language for Algebraic Computation Systems*. PhD thesis, U.C. Berkeley, EECS Dept., 1983.

**Abstract:** This thesis describes the design of a language to support a mathematics-oriented symbolic algebra system. The language, which we have named NEWSPEAK, permits the complex interrelations of mathematical types, such as rings, fields and polynomials to be described. Functions can be written over the most general type that has the required operations and properties and the inherited by subtypes. All function calls are generic, with most function resolution done at compile time. Newspeak is type-safe, yet permits runtime creation of tyhpes.

**Link:** http://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-83-160.pdf

[Fort90] Albrecht Fortenbacher. Efficient type inference and coercion in computer algebra. In *Design and Implementation of Symbolic Computation Systems*, Lecture Notes in Computer Science 429, pages 56–60, 1990, 0-387-52531-9.

**Abstract:** Computer algebra systesm of the new generation, like SCRATCHPAD, are characterized by a very rich type concept, which models the relationship between mathematical domains of computa-

tion. To use these systems interactively, however, the user should be freed of type information. A type inference mechanism determines the appropriate function to call. All known models which allow to define a semantics for type inference cannot express the rich "mathematical" type structure, so presently type inference is done heuristically. The following paper defines a semantics for a subproblem therof, namely coercion, which is based on rewrite rules. From this definition, an efficient coercion algorithm for SCRATCHPAD is constructed using graph techniques.

[Fran73] A.A. Frankel, Y. Bar-Hillel, and A. Levy. *Foundations of Set Theory*. Elsevier Science, 1973, 978-0720422702.

[Frey90] Peter Freyd and Andre Scedrov. *Categories, Allegories*. North-Holland, 1990, 978-0-444-703368-2.

**Abstract:** On the Categories side, the book centers on that part of categorical algebra that studies exactness properties, or other properties enjoyed by nice or convenient categories such as toposes, and their relationship to logic (for example, geometric logic). A major theme throughout is the possibility of representation theorems (aka completeness theorems or embedding theorems) for various categorical structures, spanning back now about five decades (as of this writing) to the original embedding theorems for abelian categories, such as the Freyd-Mitchell embedding theorem. On the Allegories side: it may be said they were first widely publicized in this book. They comprise many aspects of relational algebra corresponding to the categorical algebra studied in the first part of the book

**Comment:** Mathematical Library Vol 39

[Frue91] Thom Fruehwirth, Ehud Shapiro, Moshe Y. Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Proc. Sixth Annual IEEE Symp. on Logic in Comp. Sci.*, pages 300–309. IEEE, 1991.

**Abstract:** Type checking can be extremely useful to the program development process. Of particular interest are descriptive type systems, which let the programmer write programs without having to define or mention types. We consider here optimistic type systems for logic programs. In such systems types are conservative approximations to the success set of the program predicates. We propose the use of logic programs to describe types. We argue that this approach unifies the denotational and operational approaches to descriptive type systems and is simpler and more natural than previous approaches. We focus on the use of unary-predicate programs to describe types. We identify a proper class of unary-predicate programs and show that it is expressive enough to express several notions of types. We use an analogy with 2-way automata and a correspondence with alternating algorithms to obtain a complexity characterization of type inference and type checking. This characterization was facilitated by the use of logic programs to represent types.

[Fuhx89] You-Chin Fuh and Prateek Mishra. Polymorphic Subtype Inference – Closing the Theory-Practice Gap. *Lecture Notes in Computer Science*, 352:167–183,

1989.

[Fuhx90] You-Chin Fuh. Type Inference with Subtypes. *Theoretical Computer Science*, 73(2):155–175, 1990.

**Abstract:** We extend polymorphic type inference with a very general notion of subtype based on the concept of type transformation. This paper describes the following results. We prove the existence of (i) principal type property and (ii) syntactic completeness of the type-checker, for type inference with subtypes. This result is developed with only minimal assumptions on the underlying theory of subtypes. As a consequence, it can be used as the basis for type inference with a broad class of subtype theories. For a particular structural theory of subtypes, those engendered by inclusions between type constants only, we show that principal types are compactly expressible. This suggests that type inference for the structured theory of subtypes is feasible. We describe algorithms necessary for such a system. The main algorithm we develop is called MATCH, an extension to the classical unification algorithm. A proof of correctness for MATCH is given.

[GAPx17] The GAP Group. GAP - Reference Manual, 2017.

**Link:** https://www.gap-system.org/Manuals/doc/ref/manual.pdf

[Gath90] J. von zur Gathen and M. Giesbrecht. Constructing normal bases in finite fields. *J. Symb. Comp.*, 10:547–570, 1990.

[Gedd92] Keith Geddes, O. Czapor, Stephen R., and George Labahn. *Algorithms For Computer Algebra*. Kluwer Academic Publishers, September 1992, 0-7923-9259-0.

**Abstract:** Computer Algebra (CA) is the name given to the discipline of algebraic, rather than numerical, computation. There are a number of computer programs  Computer Algebra Systems (CASs) available for doing this. The most widely used general-purpose systems that are currently available commercially are Axiom, Derive, Macsyma, Maple, Mathematica and REDUCE. The discipline of computer algebra began in the early 1960s and the first version of REDUCE appeared in 1968. A large class of mathematical problems can be solved by using a CAS purely interactively, guided only by the user documentation. However, sophisticated use requires an understanding of the considerable amount of theory behind computer algebra, which in itself is an interesting area of constructive mathematics. For example, most systems provide some kind of programming language that allows the user to expand or modify the capabilities of the system. This book is probably the most general introduction to the theory of computer algebra that is written as a textbook that develops the subject through a smooth progression of topics. It describes not only the algorithms but also the mathematics that underlies them. The book provides an excellent starting point for the reader new to the subject, and would make an excellent text for a postgraduate or advanced undergraduate course. It is probably desirable for the reader to have some background in abstract algebra, algorithms and

programming at about second-year undergraduate level. The book introduces the necessary mathematical background as it is required for the algorithms. The authors have avoided the temptation to pursue mathematics for its own sake, and it is all sharply focused on the task of performing algebraic computation. The algorithms are presented in a pseudo-language that resembles a cross between Maple and C. They provide a good basis for actual implementations although quite a lot of work would still be required in most cases. There are no code examples in any actual programming language except in the introduction. The authors are all associated with the group that began the development of Maple. Hence, the book reflects the approach taken by Maple, but the majority of the discussion is completely independent of any actual system. The authors experience in implementing a practical CAS comes across clearly. The book focuses on the core of computer algebra. The first chapter introduces the general concept and provides a very nice historical survey. The next three chapters discuss the fundamental topics data structures, representations and the basic arithmetic of integers, rational numbers, multivariate polynomials and rational functions on which the rest of the book is built. A major technique in CA involves projection onto one or more homomorphic images, for which the ground ring is usually chosen to be a finite field. The image solution is lifted back to the original problem domain by means of the Chinese Remainder Theorem in the case of multiple homomorphic images, or the Hensel (-adic or ideal-adic) construction in the case of a single image. The next two chapters are devoted to these techniques in a fairly general setting. The two subsequent chapters specialise them to GCD computation and factorisation for multivariate polynomials; the first of these chapters also discusses the important but difficult topic of subresultants. The next two chapters describe the use of fraction-free Gaussian elimination, resultants and Grbner Bases for manipulation and exact solution of linear and nonlinear polynomial equations. The two final chapters describe "classical" algorithms and the more recent Risch algorithm for symbolic indefinite integration, and provide an introduction to differential algebra. The book does not consider more specialised problem areas such as symbolic summation, definite integration, differential equations, group theory or number theory. Nor does it consider more applied problem areas such as vectors, tensors, differential forms, special functions, geometry or statistics, even though Maple and other CASs provide facilities in all or many of these areas. It does not consider questions of CA programming language design, nor any of the important but non-algebraic facilities provided by current CASs such as their user interfaces, numerical and graphical facilities. This is a long book (nearly 600 pages); it is generally very well presented and the three authors have merged their contributions seamlessly. I noticed very few typographical errors, and none of any consequence. I have only two complaints about the book. The typeface is too small, particularly for the relatively large line spacing used, and it is much too expensive, particularly for a book that would otherwise be an ex-

cellent student text. I recommend it highly to anyone who can afford it.

[Geis89] W. Geiselmann and D. Gollmann. Symmetry and Duality in Normal Basis Multiplication. *Proc. AAECC-6, LNCS*, 357, 1989.

[Ghel91] Giorgio Ghelli. Modeling Features of Object-Oriented Languages in Second Order Functional Languages with Subtypes. *LNCS*, 489:311–340, 1991.

**Abstract:** Object oriented languages are an important tool to achieve software reusability in any kind of application and can increase dramatically software productivity in some special fields; they are also considered a logical step in the evolution of object oriented languages. But these languages lack a formal foundation, which is needed both to develop tools and as a basis for the future evolution of these languages; they lack also a strong type system, which would be essential to assure that level of reliability which is required to large evolving systems. Recently some researches have tried to insulate the basic mechanisms of object oriented languages and to embed them in strongly typed functional languages, giving to these mechanism a mathematical semantics and a set of strong type rules. This is a very promising approach which also allows a converging evolution of both the typed functional and the object oriented programming paradigms, making it possible a technology transfer in both directions. Most works in this field are very technical and deal just with one aspect of object oriented programming; many of them use a very similar framework. In this work we describe and exemplify that common framework and we survey some of the more recent and promising works on more specific features, using the framework introduced. We describe the resuks achieved and point out some problems which are still open, especially those arising from the interactions between different mechanisms.

[Gira72] Jean-Yves Girard. *Intrprétation fontionelle et élimination des coupures de l'arithmétique d'orde supérieur*. PhD thesis, Université Paris VII, 1972.

[Gira89] Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1989.

[Gode58] Kurt Godel. über eine bisher noch nicht benutzte erweiterung des finiten standpunktes, 1958.

[Gogu82] J.A. Goguen and J. Meseguer. Completeness of Many-sorted Equational Logic. *ACM SIGPLAN Notices*, 17(1):9–17, 1982.

**Abstract:** The rules of deduction which are usually used for many-sorted equational logic in computer science, for example in the study of abstract data types, are not sound. Correcting these rules by introducing explicit quantifiers yields a system which, although it is sound, is not complete; some new rules are needed for the addition and deletion of quantifiers. This note is intended as an informal, but precise, introduction to the main issues and results. It gives an example showing the unsoundness of the usual rules; it also gives a completeness theorem for our new rules, and gives necessary and sufficient conditions for the old rules to agree with the new.

[Gogu89] Joseph Goguen and Jose Meseguer. Order-sorted Algebra I : Equational Deduction for Multiple Inheritance, Overloading, Exceptions, and Partial Operations. technical report SRIR 89-10, SRI International, 1989.

[Gogu92] Joseph Goguen and Jose Meseguer. Order-sorted Algebra I : Equational Deduction for Multiple Inheritance, Overloading, Exceptions, and Partial Operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

> **Abstract:** This paper generalizes many-sorted algebra (MSA) to order-sorted algebra (OSA) by allowing a partial ordering relation on the set of sorts. This supports abstract data types with multiple inheritance (in roughly the sense of object-oriented programming), several forms of polymorphism and overloading, partial operations (as total on equationally defined subsorts), exception handling, and an operational semantics based on term rewriting. We give the basic algebraic constructions for OSA, including quotient, image, product and term algebra, and we prove their basic properties, including quotient, homomorphism, and initiality theorems. The paper's major mathematical results include a notion of OSA deduction, a completeness theorem for it, and an OSA Birkhoff variety theorem. We also develop conditional OSA, including initiality, completeness, and McKinsey-Malcev quasivariety theorems, and we reduce OSA to (conditional) MSA, which allows lifting many known MSA results to OSA. Retracts, which intuitively are left inverses to subsort inclusions, provide relatively inexpensive run-time error handling. We show that it is safe to add retracts to any OSA signature, in the sense that it gives rise to a conservative extension. A final section compares and contrasts many different approaches to OSA. This paper also includes several examples demonstrating the flexibility and applicability of OSA, including some standard benchmarks like stack and list, as well as a much more substantial example, the number hierarchy from the naturals up to the quaternions.

[Gold83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[Gonz89] Laureano Gonzalex, Lombardi Henri, Tomas Recio, and Marie-Francoise Roy. Sturm-Habicht sequence. In *Proc. ACM-SIGSAM 1989*, pages 136–146, 1989, 0-89791-325-6.

> **Abstract:** Formal computations with inequalities is a subject of general interest in computer algebra. In particular it is fundamental in the parallelisation of basic algorithms and quantifier elimination for real closed filed ([BKR], [HRS]). In §$I$ we give a generalisation of Sturm theorem essentially due to Sylvester which is the key for formal computations with inequalities. Our result is an improvement of previously known results (see [BKR]) since no hypotheses have to be made on polynomials. In §$II$ we study the subresultant sequence. We precise some of the classical definitions in order to avoid some problems appearing in the paper by Loos ([L]) and study specialisation properties in detail. In §$III$ we introduce the Sturm-Habicht sequence, which generalises Habicht's work ([H]). This new sequence obtained automatically from a subresultant sequence has some re-

markable properties:

- it gives the same information than the Sturm sequence, and this information may be recovered by looking only at its principal coefficients
- it can be computed by ring operations and exact divisions only, in polynomial time in case of integer coefficients, eventually by modular methods
- it has goos specialisation properties

Finally in §*IV* we give some information about applications and implementation of the Sturm-Habicht sequence.

[Gonz98] L. Gonzalex-Vega. A combinatorial algorithm solving some quantifier elimination problems. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 365–374, 1998, 3-211-82794-3.

[Grae79] George Graetzer. *Universal Algebra*. Springer, 1979, 978-0-387-77486-2.

[Grie78] David Gries. *Programming Methodology*. Springer-Verlag, 1978.

[Grig88] D. Yu. Grigor'ev and N. N. Vorobjov. Solving systems of polynomial inequalities in subexponential time. *J. Symbolic Computation*, 5(1-2):37–64, 1988.

**Abstract:** Let the polynomials $f_1, \ldots, f_k \in \mathbb{Z}[X_1, \ldots, X_n]$ have degrees $deg(f_i) < d$ and absolute value of any coefficient of less than or equal to $s^M$ for all $1 \leq i \leq k$. We describe an algorithm which recognizes the existence of a real solution of the system of inequalities $f_1 > 0, \ldots, f_k \geq 0$. In the case of a positive answer the algorithm constructs a certain finite set of solutions (which is, in fact, a representative set for the family of components of connectivity of the set of all real solutions of the system). The algorithm runs in time polynomial in $M(kd)^{n^2}$. The previously known upper time bound for this problem was $(Mkd)^{2^{O(n)}}$.

[Grig88a] D. Yu. Grigor'ev. The complexity of deciding Tarski algebra. *J. Symbolic Computation*, 5(1-2):65–108, 1988.

**Abstract:** Let a formula of Tarski algebra contain $k$ atomic subformulas of the kind $(f_i \geq 0)$, $1 \leq i \leq k$, where the polynomials $f_i \in \mathbb{Z}[X_1, \ldots, X_n]$ have degrees $deg(f_i) < d$, let $2^M$ be an upper bound for the absolute value of every coefficient of the polynomials $f_i$, $1 \leq i \leq k$, let $a \leq n$ be the number of quantifier alternations in the prenex form of the formula. A decision method for Tarski algebra is described with the running time polynomial in $M(kd)^{(O(n))^{4a-2}}$. Previously known decision procedures have a time complexity polynomial in $(Mkd)^{2^{O(n)}}$

[Grog91] Peter Grogono. Issues in the Design of an Object Oriented Programming Language, 1991.

**Abstract:** The object oriented paradigm, which advocates bottom-up program development, appears at first sight to run counter to the classical, top-down approach of structured programming. The deep requirement of structured programming, however, is that program-

ming should be based on well-defined abstractions with clear meaning rather than on incidental characteristics of computing machinery. This requirement can be met by object oriented programming and, in fact, object oriented programs may have better structure than programs obtained by functional decomposition. The definitions of the basic components of object oriented programming, object, class, and inheritance, are still sufficiently fluid to provide many choices for the designer of an object oriented language. Full support of objects in a typed language requires a number of features, including classes, inheritance, genericity, renaming, and redefinition. Although each of these features is simple in itself, interactions between features can become complex. For example, renaming and redefinition may interact in unexpected ways. In this paper, we show that appropriate design choices can yield a language that fulfills the promise of object oriented programming without sacrificing the requirements of structured programming.

**Comment:** Published in Structured Programming

**Link:** http://users.encs.concordia.ca/~grogono/Writings/oopissue.pdf

[Hach95] G. Haché and D. Le Brigand. Effective construction of algebraic geometry codes. *IEEE Transaction on Information Theory*, 41(6):1615–1628, November 1995.

**Abstract:** We intend to show that algebraic geometry codes (AG-codes, introduced by Goppa in 1977 [5]) can be constructed easily using blowing-up theory. This work is based on a paper by Le Brigand and Risler. Given a plane curve, we desingularize the curve by means of blowing-up, and then using the desingularisation trees and the monoidal transformations associated to the blowing-up morphisms, we compute the adjoint divisor of the curve. Finally we show how to use the algorithm of Brill-Noether to compute a basis of the vector space associated to a divisor of the curve. Two examples of constructions of AG-codes are given at the end.

**Link:** https://hal.inria.fr/inria-00074404/file/RR-2267.pdf

**Algebra:**
(p**??**) package GPAFF GeneralPackageForAlgebraicFunctionField
(p**??**) package PAFFFF PackageForAlgebraicFunctionFieldOverFiniteField
(p**??**) package PAFF PackageForAlgebraicFunctionField

[Hach96] G. Haché. *Construction effective des codes géométriques.* PhD thesis, l'Université Pierre et Marie Curie (Paris 6), Septembre 1996.

[Hamm62] R W. Hamming. *Numerical Methods for Scientists and Engineers.* Dover, 1973, 0-486-65241-6.

[Harp89] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-Order Modules and the Phase Distinction. In *Symp. on Principles of Programming Languages POPL'17*, pages 341–354. ACM Press, 1989.

**Abstract:** In earlier work, we used a typed function calculus, XML, with dependent types to analyze several aspects of the Standard ML type system. In this paper , we introduce a refinement of XML with a clear compile-time/run-time phase distinction, and a direct compile-

time type checking algorithm. The calculus uses a finer separation of types into universes than XML and enforces the phase distinction using a nonstandard equational theory for module and signature expressions. While unusual from a type-theoretic point of view, the nonstandard equational theory arises naturally from the well-known Grothendieck construction on an indexed category.

**Comment:** CMU-CS-89-198

**Link:** https://www.cs.cmu.edu/~rwh/papers/phase/tr.pdf

[Harp93] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *J. ACM*, 40(1):143–184, 1993.

**Abstract:** The Edinburgh Logical Framework (LF) provides a means to define (or present) logics. It is based on a general treatment of syntax, rules, and proofs by means of a typed $\lambda$-calculus with dependent types. Syntax is treated in a style similar to, but more general than, Martin-Lof's system of arities. The treatment of rules and proofs focuses on his notion of a judgment. Logics are represented in LF via a new principle, the judgments as types principle, whereby each judgment is identified with the type of its proofs. This allows for a smooth treatment of discharge and variable occurence conditions and leads to a uniform treatment of rules and proofs whereby rules are viewed as proofs of higher-order judgments and proof checking is reduced to type checking. The practical benefit of our treatment of formal systems is that logic-independent tools, such as proof editors and proof checkers, can be constructed.

[Harp93a] Robert Harper and John C. Mitchell. On the Type Structure of Standard ML. *Transactions on Programming Languages and Systems*, 15(2):211–252, 1993.

**Abstract:** Standard ML is a useful programming module facility. One notable feature of the core expression language of ML is that it is implictly typed: no explicit type information need be supplied by the programmer. In contrast, the module language of ML is explicitly typed; in particular, the types of parameters in parametric modules must be supplied by the programmer. We study the type structure of Standard ML by giving an explicitly-typed, polymmorphic function calculus that captures many of the essential aspects of both the core and module language. In this setting, implicitly-type core language expressions are reguarded as a convenient short-hand for an explicitly-typed counterpart in our function calculus. In contrast to the Girard-Reynolds polymorphic calculus, our function calculus is predicative: the type system may be built up by induction on type levels. We show that, in a precise sense, the language becomes inconsistent if restrictions imposed by type levels are relaxed. More specifically, we prove that the important programming features of ML cannot be added to any impredicative language, such as the Girard-Reynolds calcus, without implicitly assuming a type of all types.

[Harp94] Robert Harper and Mark Lillibridge. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *POPL'21 Principles of Programming Languages*. ACM Press, 1994.

**Abstract:** The design of a module system for constructing and maintaining large programs is a difficult task that raises a number of theoretical and practical issues. A fundamental issue is the management of the flow of information between program units at compile time via the notion of an interface. Experience has shown that fully opaque interfaces are awkward to use in practice since too much information is hidden, and that fully transparent interfaces lead to excessive interdependencies, creating problems for maintenance and separate compilation. The "sharing" specications of Standard ML address this issue by allowing the programmer to specify equational relationships between types in separate modules, but are not expressive enough to allow the programmer complete control over the propagation of type information between modules. These problems are addressed from a type-theoretic viewpoint by considering a calculus based on Girard's system $F_\omega$. The calculus differs from those considered in previous studies by relying exclusively on a new form of weak sum type to propagate information at compile-time, in contrast to approaches based on strong sums which rely on substitution. The new form of sum type allows for the specification of equational, as well as type and kind, information in interfaces. This provides complete control over the propagation of compile-time information between program units and is sufficient to encode in a straightforward way most uses of type sharing specifications in Standard ML. Modules are treated as "first-class" citizens, and therefore the system supports higher-order modules and some object-oriented programming idioms: the language may be easily restricted to "second-class' modules found in ML-like languages.

[Hein89] J. Heintz, M-F. Roy, and P. Solerno. On the complexity of semialgebraic sets. In *Proc. IFIP*, pages 293–298, 1989.

[Herm1872] E. Hermite. Sur l'intégration des fractions rationelles, 1872.

[Herr73] Horst Herrlich and G.E. Strecker. *Category Theory: An Introduction.* Allyn and Bacon, 1973.

[Hind69] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. AMS*, 146:29–60, 1969.

[Hodg95] Wilfrid Hodges. The Meaning of Specifications I: Domains and Initial Models. *Theoretical Computer Science*, 192:67–89, 1995.

**Abstract:** This is the first of a short series of papers intended to provide one common semantics for several different types of specification language, in order to allow comparison and translations. The underlying idea is that a specification describes the behaviour of a system, depending on parameters. We can represent this behaviour as a functor which acts on structures representing the parameters, and which yields a structure representing the behaviour. We characterise in domain-theoretic terms the class of functors which could in principle be specified and implemented; briefly, they are the functors which preserve directed colimits and whose restriction to finitely presented structures is recursively enumerable. We also characterise those func-

tors which allow specification by initial semantics in universal Horn classes with finite vocabulary; these functors consist of a free functor (i.e. left adjoint of a forgetful functor) followed by a forgetful functor. The main result is that these two classes of functor are the same up to natural isomorphism.

[Hoei94] M. van Hoeij. An algorithm for computing an integral basis in an algebraic function field. *Journal of Symbolic Computation*, 18(4):353–363, 1994.

**Abstract:** Algorithms for computing integral bases of an algebraic function field are implemented in some computer algebra systems. They are used e.g. for the integration of algebraic functions. The method used by Maple 5.2 and AXIOM is given by Trager in [Trag84]. He adapted an algorithm of Ford and Zassenhaus [Ford, 1978], that computes the ring of integers in an algebraic number field, to the case of a function field. It turns out that using algebraic geometry one can write a faster algorithm. The method we will give is based on Puiseux expansions. One cas see this as a variant on the Coates' algorithm as it is described in [Davenport, 1981]. Some difficulties in computing with Puiseux expansions can be avoided using a sharp bound for the number of terms required which will be given in Section 3. In Section 5 we derive which denominator is needed in the integral basis. Using this result 'intermediate expression swell' can be avoided. The Puiseux expansions generally introduce algebraic extensions. These extensions will not appear in the resulting integral basis.

[Hong05] Hoon Hong. Tutorial on CAD, 2005.

[Hong90] Hoon Hong. An improvement of the projection operator in cylindrical algebraic decomposition. In *ISSAC'90*, pages 261–264. ACM, 1990.

**Abstract:** The cylindrical algebraic decomposition (CAD) of Collins (1975) provides a potentially powerful method for solving many important mathematical problems, provided that the required amount of computation can be sufficiently reduced. An important component of the CAD method is the projection operation. Given a set $A$ of $r$-variate polynomials, the projection operation produces a certain set $P$ of $(rl)$-variate polynomials such that a CAD of $r$-dimensional space for $A$ can be constructed from a CAD of $(r1)$-dimensional space for $P$. The CAD algorithm begins by applying the projection operation repeatedly, beginning with the input polynomials, until univariate polynomials are obtained. This process is called the projection phase.

[Hong90a] Hoon Hong. *Improvements in CAD-based Quantifier Elimination*. PhD thesis, Ohio State University, 1990.

**Abstract:** Many important mathematical and applied mathematical problems can be formulated as quantifier elimination problems (QE) in elementary algebra and geometry. Among several proposed QE methods, the best one is the CAD-based method due to G. E. Collins. The major contributions of this thesis are centered around improving each phase of the CAD-based method: the projection phase, the truth-invariant CAD construction phase, and the solution formula construction phase Improvements in the projection phase. By

generalizing a lemma on which the proof of the original projection operation is based, we are able to reduce the size of the projection set significantly, and thus speed up the whole QE process. Improvements in the truth-invariant CAD construction phase. By intermingling the stack constructions with the truth evaluations, we are usually able to complete quantifier elimination with only a partially built CAD, resulting in significant speedup. Improvements in the solution formula construction phase. By constructing defining formulas for a collection of cells instead of individual cells, we are often able to produce simple solution formulas, without incurring the enormous cost of augmented projection. The new CAD-based QE algorithm, which integrates all the improvements above, has been implemented and tested on ten QE problems from diverse application areas. The overall speedups range from 2 to perhaps 300,000 times at least We also implemented D. Lazard's recent improvement on the projection phase. Tests on the ten QE problems show additional speedups by up to 1.8 times.

[Hong93] Hoon Hong. Quantifier elimination for formulas constrained by quadratic equations. In *Proc. ISSAC'93*, pages 264–274. ACM, 1993, 0-89791-604-2.

**Abstract:** An algorithm is given for constructing a quantifier free formula (a boolean expression of polynomial equations and inequalitier) equivalent to a given formula of the form: $(\exists x \in \mathbb{R})[a_x^2 + a_1 x + a_0 = 0 \wedge F]$, where $F$ is a quantifier free formula in $x_1, \ldots, x_r, x$, and $a_2, a_1, a_0$ are polynomials in $x_1, \ldots, x_r$ with real coefficients such that the system $\{a_2 = 0, a_1 = 0, a_0 = 0\}$ has no solution in $\mathbb{R}^r$. Formulas of this form frequently occur in the context of constraint logic programming over the real numbers. The output formulas are made of resultants and two variants, which we call *trace* and *slope* resultants. Both of these variant resultants can be expressed as determinants of certain matrices.

[Hong93a] Hoon Hong. Quantifier Elimination for Formulas Constrained by Quadratic Equations via Slope Resultants. *The Computer Journal*, 36(5):439–449, 1993.

**Abstract:** An algorithm is given for eliminating the quantifier from a formula $(\exists x \in \mathbb{R})[a_x^2 + a_1 x + a_0 = 0 \wedge F]$, where $F$ is a quantifier free formula in $x_1, \ldots, x_r, x$ and $a_2, a_1, a_0$ are polynomials in $x_1, \ldots, x_r$ with real coefficients such that the system $\{a_2 = 0, a_1 = 0, a_0 = 0\}$ has no solution in $\mathbb{R}^r$. The output formulas are made of resultants and their variants, which we call *slope* resultants. The slope resultants can be, like the resultants, expressed as determinants of certain matrices. If we allow the determinant symbol in the output the computing time of the algorithm is *linear* in the length of the input. If not, the computing time is dominated by $N(n^{2r+1}\ell + n^{2r}\ell^2)$ where $N$ is the number of polynomials in the input formula, $r$ is the number of variables, $n$ is the maximum of the degrees for every variable, and $\ell$ is the maximum of the integer coefficient bit lengths. Experimants with implementation suggest that the algorithm is sufficiently efficient to be useful in practice.

[Hong94] Hoon Hong and V. Stahl. Safe start regions by fixed points and tightening. *Computing*, 53(3):323–335, 1994.

**Abstract:** In this paper, we present a method for finding safe starting regions for a given system of non-linear equations. The method is an improvement of the usual method which is based on the fixed point theorem. The improvement is obtained by enclosing the components of the equation system by univariante interval polynomials whose zero sets are found. This operation is called "ightening". Preliminary experiments show that the tightening operation usually reduces the number of bisections, and thus the computing time. The reduction seems to become more dramatic when the number of variables increases.

[Hong98] Hoon Hong. Simple Solution Formula Construction in Cylindrical Algebraic Decomposition Based Quantifier Elimination. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 201–219. Springer, 1998, 3-211-82794-3.

**Abstract:** In this paper, we present several improvements to the last step (solution formula construction step) of Collins' cylindrical algebraic decomposition based quantifier elimination algorithm. The main improvements are

- that it does *not* use the expensive augmented projection used by Collins' original algorithm, and
- that it produces *simple* solution formulas by using three-valued logic minimization

For example, for the quadratic problem studied by Arnon, Mignotte, and Lazard, the solution formula produced by the original algorithm consists of 401 atomic formulas, but that by the improved algorithm consists of 5 atomic formulas.

**Comment:** also ISSAC'92 pages 177-188, 1992

[Hong98a] Hoon Hong and J. Rafael Sendra. Computation of Variant Results. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 327–337. Springer, 1998, 3-211-82794-3.

[Howe87] Douglas J. Howe. The Computational Behaviour of Girard's Paradox. Technical Report TR 87-820, Cornell University, 1987.

**Abstract:** In their paper "Type" Is Not a Type, Meyer and Reinhold argued that serious pathologies can result when a type of all types is added to a programing language with dependent types. Central to their argument is the claim that by following the proof of Girard's paradox it is possible to construct in their calculus $\lambda^{\tau\tau}$ a term having a fixed-point property. Because of the tremendous amount of formal detail involved, they were unable to establish this claim. We have made use of the Nuprl proof development system in constructing a formal proof of Girard's paradox and analysing the resulting term. We can show that the term does not have the desired fixed-point property, but does have a weaker form of it that is sufficient to establish some of the results of Meyer and Reinhold. We believe that the method used here is in itself of some interest, representing a new kind of application of a computer to a problem in symbolic logic.

**Link:** https://ecommons.cornell.edu/handle/1813/6660

[Hube90] K. Huber. Some Comments on Zech's Logarithm. *IEEE Trans. Information Theory*, IT-36:946–950, 1990.

[Huda92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, Maria M. Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Patrain, and John Peterson. Report on the Programming Language Haskell, a non-strict functional language version 1.2. *ACM SIGPLAN Notices*, 27(5):1–164, 1992.

**Abstract:** Some half dozen persons have written technically on combinatory logic, and most of these, including ourselves, have published something erroneous. Since some of our fellow sinners are among the most careful and competent logicians on the contemporary scene, we regard this as evidence that the subject is refractory. Thus fullness of exposition is necessory for accurary; and excessive condensation would be false economy here, even more than it is ordinarily.

[Huda99] Paul Hudak, John Peterson, and Joseph H. Fasel. A Gentle Introduction to Haskell 98, 1999.

**Link:** https://www.haskell.org/tutorial/haskell-98-tutorial.pdf

[Huet91] Gerard Huet and G. Plotkin. *Logical Frameworks*. Cambridge University, 1991.

[Itoh88] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Inf. and Comp.*, 78:171–177, 1988.

**Abstract:** This paper proposes a fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. Normal bases have the following useful property: In the case that an element $x$ in $GF(2^m)$ is represented by normal bases, $2^k$ power operation of an element $x$ in $GF(2^m)$ can be carried out by $k$ times cyclic shift of its vector representation. C.C. Wang et al. proposed an algorithm for computing multiplicative inverses using normal bases, which requires $(m-2)$ multiplications in $GF(2^m)$ and $(m-1)$ cyclic shifts. The fast algorithm proposed in this paper also uses normal bases, and computes multiplicative inverses iterating multiplications in $GF(2^m)$. It requires at most $2[log_2(m-1)]$ multiplications in $GF(2^m)$ and $(m-1)$ cyclic shifts, which are much less than those required in Wang's method. The same idea of the proposed fast algorithm is applicable to the general power operation in $GF(2^m)$ and the computation of multiplicative inverses in $GF(q^m)$ $(q = 2^n)$.

**Algebra:**
(p**??**) package INBFF InnerNormalBasisFieldFunctions

[Jaco85] N. Jacobson. *Basic Algebra I, 2nd ed.* W.H. Freeman and Co., 1985.

[Jaes91] G. Jaeschke. Private Communication, 1991.

**Comment:** to James Davenport

[Jenk84b] Richard D. Jenks. A primer: 11 keys to New Scratchpad. In *Proc. EUROSAM ISSAC 1984*, pages 123–147. Springer-Verlag, 1984, 0-387-13350-X.

**Abstract:** This paper is an abbreviated primer for the language of new SCRATCHPAD, a new implementation of SCRATCHPAD which has been under design and development by the Computer Algebra

Group at the IBM Research Center during the past 6 years. The basic design goals of the new SCRATCHPAD language and interface to the user are to provde:

- a "typeless" interactive language suitable for on-line solution of mathematical problems by novice users with little or no programming required, and
- a programming language suitable for the formal description of algorithms and algebraic structures which can be compiled into run-time efficient object code.

The new SCRATCHPAD language is introduced by 11 keys with each successive key introducing an additional capability of the language. The language is thus described as a "concentric" language with each of the 11 levels corresponding to a language subset. These levels are more than just a pedagogic device, since they correspond to levels at which the system can be effectively used. Level 1 is sufficient for naive interactive use; levels 2-8 progressively introduce interactive users to capabilities of the language; levels 9-11 are for system programmers and advanced users. Levesl 2, 4, 6, and 7 give users the full power of LISP with a high-level language; level 8 introduces "type declarations;" level 9 allows polymorphic functions to be defined and compiled; levels 10-11 give users an Ada-like facility for defining types and packages (those of new SCRATCHPAD are dynamically constructable, however). One language is used for both interactive and system programming language use, although several freedoms such as abbreviation and optional type-declarations allowed at top-level are not permitted in system code. The interactive language (levels 1-8) is a blend of original SCRATCHPAD [GRJY75], some proposed extensions [JENK74], work by Loos [LOOS74], SETL [DEWA79], SMP [COWO81], and new ideas; the system programming language (levels 1-11) superficially resembles Ada but is more similar to CLU [LISK74] in its semantic design. This presentation of the language in this paper omits many details to be covered in the SCRATCHPAD System Programming Manual [SCRA84] and an expanded version of this paper will serve as a primer for SCRATCHPAD users [JESU84].

[Jenk92] Richard D. Jenks and Robert S. Sutor. *AXIOM: The Scientific Computation System*. Springer-Verlag, Berlin, Germany, 1992, 0-387-97855-0.

[Jone87] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Simon and Schuster, 1987, 0-13-453333-X.

[Joua90] Jean-Pierre Jouannaud and Claude Kirchner. *Solving Equations in Abstract Algebras: A Rule-based Survey of Unification*. Universite do Paris-Sud, 1990.

[Joua91] Jean Pierre Jouannaud and Mitsuhiro Okada. A Computation Model for Executable Higher-order Algebraic Specification Languages. In *Symposium on Logic in Computer Science*, pages 350–361, 1991, 081862230X.

**Abstract:** The combination of (polymorphically) typed lambda-calculi with first-order as well as higher-order rewrite rules is considered. The need of such a combination for exploiting the benefits of algebraically defined data types within functional programming is

demonstrated. A general modularity result, which allows as particular cases primitive recursive functionals of higher types, transfinite recursion of higher types, and inheritance for all types, is proved. The class of languages considered is first defined, and it is shown how to reduce the Church-Rosser and termination (also called strong normalization) properties of an algebraic functional language to a so-called principal lemma whose proof depends on the property to be proved and on the language considered. The proof of the principal lemma is then sketched for various languages. The results allows higher order rules defining the higher-order constants by a certain generalization of primitive recursion. A prototype of such primitive recursive definitions is provided by the definition of the map function for lists.

[Kaes92] Stefan Kaes. Type Inference inthe Presence of Overloading, Subtyping and Recursive Types. *ACM Lisp Pointers*, 5(1):193–204, 1992.

**Abstract:** We present a unified approach to type inference in the presence of overloading and coercions based on the concept of *constrained types*. We define a generic inference system, show that subtyping and overloading can be treated as a special instance of this system and develop a simple algorithm to compute principal types. We prove the decidability of type inference for hte class of *decomposable predicates* and deelop a canonical representation for principal types based on *most accurate simplifications* of constraint sets. Finally, we investigate the extension of our techniques to *recursive types*.

[Kalt83a] E. Kaltofen. Factorization of Polynomials. In *Computer Algebra - Symbolic and Algebraic Computation*, pages 95–113. ACM, 1983.

**Abstract:** Algorithms for factoring polynomials in one or more variables over various coefficient domains are discussed. Special emphasis is given to finite fields, the integers, or algebraic extensions of the rationals, and to multivariate polynomials with integral coefficients. In particular, various squarefree decomposition algorithms and Hensel lifting techniques are analyzed. An attempt is made to establish a complete historic trace for todays methods. The exponential worst case complexity nature of these algorithms receives attention.

[Kane90] Paris C. Kanellakis, Harry G. Mairson, and John C. Mitchell. Unification and ML Type Reconstruction. Technical Report CS-90-26, Brown University, 1990.

**Abstract:** We study the complexity of type reconstruction for a core fragment of ML with lambda abstraction, function application, and the polymorphic **let** declaration. We derive exponential upper and lower bounds on recognizing the typable core ML expressions. Our primary technical tool is unification of succinctly represented type expressions. After observing that core ML expressions, of size $n$, can be typed in DTIME($s^n$), we exhibit two different families of programs whose principal types grow exponentially. We show how to exploit the expressiveness of the **let**-polymorphism in these constructions to derive lower bounds on deciding typability: one leads naturally to NP-hardness and the other to DTIME($2^{n^k}$)-hardness for each integer $k \geq 1$. Our generic simulation of any exponential time Turing Ma-

chine by ML type reconstruction may be viewed as a nonstandard way of computing with types. Our worse-case lower bounds stand in contrast to practical experience, which suggests that commonly used algorithms for type reconstruction do not slow compilation substantially.

**Link:** ftp://ftp.cs.brown.edu/pub/techreports/90/cs90-26.pdf

[Kfou88] A.J. Kfoury, J. Tiuryn, and P. Utzyczyn. A Proper Extension of ML with an Effective Type-Assignment. In *POPL 88*, pages 58–69, 1988.

**Abstract:** We extend the functional language ML by allowing the recursive calls to a function F on the right-hand side of its definition to be at different types, all generic instances of the (derived) type of F on the left-hand side of its definition. The original definition of ML does not allow this feature. This extension does not produce new types beyond the usual universal polymorphic types of ML and satisfies the properties already enjoyed by ML: the principal-type property and the effective type-assignment property.

[Kfou93] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The Undecidability of the Semi-unification Problem. *Information and Computation*, 102(1):83–101, 1993.

**Abstract:** The Semi-Unification Problem (SUP) is a natural generalization of both first-order unification and matching. The problem arises in various branches of computer science and logic. Although several special cases of SUP are known to be decidable, the problem in general has been open for several years. We show that SUP in general is undecidable, by reducing what we call the "boundedness problem" of Turing machines to SUP. The undecidability of this boundedness problem is established by a technique developed in the mid-1960s to prove related results about Turing machines.

[Kife91] Michael Kifer and James Wu. A First-order Theory of Types and Polymorphism in Logic Programming. In *Proc Sixth Annual IEEE Symp. on Logic in Comp. Sci.*, pages 310–321, 1991.

**Abstract:** A logic called typed predicate calculus (TPC) that gives declarative meaning to logic programs with type declarations and type inference is introduced. The proper interaction between parametric and inclusion varieties of polymorphism is achieved through a construct called type dependency, which is analogous to implication types but yields more natural and succinct specifications. Unlike other proposals where typing has extra-logical status, in TPC the notion of type-correctness has precise model-theoretic meaning that is independent of any specific type-checking or type-inference procedure. Moreover, many different approaches to typing that were proposed in the past can be studied and compared within the framework of TPC. Another novel feature of TPC is its reflexivity with respect to type declarations; in TPC, these declarations can be queried the same way as any other data. Type reflexivity is useful for browsing knowledge bases and, potentially, for debugging logic programs.

[Kirk89] Bjorn Kirkerud. *Object-Oriented Programming With Simula*. International Computer Science Series. Addison-Wesley, 1989.

[Klop90] J. W. Klop. Term Rewriting Systems. Technical Report CS-R9073, Stichting Methematisch Centrum, 1990.

**Abstract:** Term Rewriting Systems play an important role in various areas, such as abstract data type specifications, implementations of functional programming languages and automated deduction. In this chapter we introduce several of the basic concepts and facts for TRSs. Specifically, we discuss Abstract Reduction Systems; general Term Rewriting Systems including an account of Knuth-Bendix completion and (E- )unification; orthogonal TRSs and reduction strategies; strongly sequential orthogonal TRS. Finally some extended rewrite formates are introduced: Conditional TRSs and Combinatory Reduction Systems. The emphasis throughout the paper is on providing information of a syntactic nature.

[Knut71] Donald Knuth. *The Art of Computer Programming Vol. 2 (Seminumerical Algorithms)*. Addison-Wesley, 1971.

[Kobl87] N. Koblitz. *A Cource in Number Theory and Cryptography*. Springer-Verlog, 1987.

[Koun90] Emmanuel Kounalis and Michael Rusinowitch. A Proof System for Conditional Algebraic Specifications. In *Conference Conditional and Typed Rewriting Systems*, 1990.

**Abstract:** Algebraic specifications provide a formal basis for designing data-structures and reasoning about their properties. Sufficient-completeness and consistency are fundamental notions for building algebraic specifications in a modular way. We give in this paper effective methods for testing these properties for specifications with conditional axioms.

[Kowa63] Hans Joachim Kowalsky. *Linear Algebra*. Walter de Gruyter, 1963.

**Comment:** (German)

[Lamb06] Branimir Lambov. Interval Arithmetic Using SSE-2. In *Lecture Notes in Computer Science*, pages 102–113. Springer-Verlag, 2006, 978-3-540-85520-0.

[Lamp88] B. Lampson and R. Burstall. Pebble, a Kernel Language for Modules and Abstract Data Types. *Information and Computation*, 76:278–346, 1988.

**Abstract:** A small set of constructs can simulate a wide variety of apparently distinct features in modern programming languages. Using a kernel language called Pebble based on the typed lambda calculus with bindings, declarations, dependent types, and types as compile time values, we show how to build modules, interfaces and implementations, abstract data types, generic types, recursive types, and unions. Pebble has a concise operational semantics given by inference rules.

[Lang05] Serge Lang. *Algebra*. Graduate Texts in Mathematics. Springer, 2005, 978-0387953854.

[Laue82] M. Lauer. Computing by Homomorphic Images. In *Computer Algebra: Symbolic and Algebraic Computation*, pages 139–168. Springer, 1982, 978-3-211-81684-4.

**Abstract:** After explaining the general technique of Computing

by homomorphic images, the Chinese remainder algorithm and the Hensel lifting construction are treated extensively. Chinese remaindering is first presented in an abstract setting. Then the specialization to Euclidean domains, in particular $\mathbb{Z}$, $\mathbb{K}[y]$, and $\mathbb{Z}[y_1, \ldots, y_n]$ is considered. For both techniques, Chinese remaindering as well as the lifting algorithms, a complete computational example is presented and the most frequent application is discussed.

[Laza90] Daniel Lazard and Renaud Rioboo. Integration of rational functions: Rational computation of the logarithmic part. *Journal of Symbolic Computation*, 9(2):113–115, February 1990.

**Abstract:** A new formula is given for the logarithmic part of the integral of a rational function, one that strongly improves previous algorithms and does not need any computation in an algebraic extension of the field of constants, nor any factorisation since only polynomial arithmetic and GCD computations are used. This formula was independently found and implemented in SCRATCHPAD by B.M. Trager.

[LeBr88] J.J. Le Brigand, D.; Risler. Algorithme de Brill-Noether et codes de Goppa. *Bull. Soc. Math. France*, 116:231–253, 1988.

[Leec92] J. Leech. Private Communication, 1992.

**Comment:** to James Davenport

[Leis87] Hans Leiss. On Type Inference for Object-Oriented Programming Languages. In *Int. Workshop on Computer Science Logic*, pages 151–172, 1987.

**Abstract:** We present a type inference calculus for object-oriented programming languages. Explicit polymorphic types, subtypes and multiple inheritance are allowed. Class types are obtained by selection from record types, but not considered subtypes of record types. The subtype relation for class types reflects the (mathematically clean) properties of subclass relations in object-oriented programming to a better extend than previous systems did. Based on Mitchells models for type inference, a semantics for types is given where types are sets of values in a model of type-free lambda calculus. For the sublanguage without type quantifiers and subtype relation, automatic type inference is possible by extending Milners algorithm W to deal with a polymorphic fixed-point rule.

[Lens82] A.K. Lenstra. Factorization of Polynomials, Comp. Methods in Number Theory (part 1). *Math. Centre Tracts*, 154, 1982.

[Lens87] H. W. Lenstra and R. J. Schoof. Primitive Normal Bases for Finite Fields. *Mathematics of Computation*, 48(177):217–231, 1987.

**Abstract:** It is proved that any finite extension of a finite field has a normal basis consisting of primitive roots

**Link:** http://www.math.leidenuniv.nl/~hwl/PUBLICATIONS/

**Algebra:**
(p**??**) package FFPOLY FiniteFieldPolynomialPackage

[Lens91] H.W. Lenstra Jr. Finding Isomorphisms between Finite Fields. *Math. of Comp.*,

56(193):329–347, 1991.

[Lero17] Xavier Leroy, Damien Doligez, Alain Frish, Jacques Garrigue, Didier Remy, and Jerome Vouillon. *The OCaml System (release 4.06)*. INRIA, 2017.

**Link:** https://caml.inria.fr/pub/distrib/ocaml-4.06/ocaml-4.06-refman.pdf

[Lero94] Xavier Leroy. Manifest Types, Modules, and Separate Compilation. In *Principles of Programming Languages POPL'94*, pages 109–122. ACM Press, 1994.

**Abstract:** This paper presents a variant of the SML module system that introduces a strict distinction between abstract types and manifest types (types whose definitions are part of the module specification), while retaining most of the expressive power of the SML module system. The resulting module system provides much better support for separate compilation.

[Lidl83] Rudolf Lidl and Harald Niederreiter. *Finite Field, Encyclopedia of Mathematics and Its Applications*, volume 20. Cambridge Univ. Press, 1983, 0-521-30240-4.
**Algebra:**
(p**??**) category FAXF FiniteAlgebraicExtensionField
(p**??**) domain FF FiniteField
(p**??**) domain FFCG FiniteFieldCyclicGroup
(p**??**) domain FFCGX FiniteFieldCyclicGroupExtension
(p**??**) domain FFCGP FiniteFieldCyclicGroupExtensionByPolynomial
(p**??**) domain FFX FiniteFieldExtension
(p**??**) domain FFP FiniteFieldExtensionByPolynomial
(p**??**) domain FFNB FiniteFieldNormalBasis
(p**??**) domain FFNBX FiniteFieldNormalBasisExtension
(p**??**) domain FFNBP FiniteFieldNormalBasisExtensionByPolynomial
(p**??**) domain IFF InnerFiniteField
(p**??**) domain IPF InnerPrimeField
(p**??**) domain PF PrimeField
(p**??**) package INBFF InnerNormalBasisFieldFunctions
(p**??**) package FFPOLY2 FiniteFieldPolynomialPackage2
(p**??**) package FFPOLY FiniteFieldPolynomialPackage
(p**??**) package FFHOM FiniteFieldHomomorphisms
(p**??**) package FFF FiniteFieldFunctions

[Limo92] C. Limongelli and M. Temperini. Abstract Specification of Structures and Methods in Symbolic Mathematical Computation. *Theoretical Computer Science*, 104:89–107, 1992.

**Abstract:** This paper describes a methodology based on the object-oriented programming paradigm, to support the design and implementation of a symbolic computation system. The requirements of the system are related to the specification and treatment of mathematical structures. This treatment is considered from both the numerical and the symbolic points of view. The resulting programming system should be able to support the formal definition of mathematical data structures and methods at their highest level of abstraction, to perform computations on instances created from such definitions, and to handle abstract data structures through the manipulation of their logical properties. Particular consideration is given to the correctness

aspects. Some examples of convenient application of the proposed design methodology are presented.

[Linc92] Patrick Lincoln and John C. Mitchell. Algorithmic Aspects of Type Inference with Subtypes. In *POPL 92*, pages 293–304, 1992.

**Abstract:** We study the complexity of type inference for programming languages with subtypes. There are three language variations that effect the problem: (i) basic functions may have polymorphic or more limited types, (ii) the subtype hierarchy may be fixed or vary as a result of subtype declarations within a program, and (iii) the subtype hierarchy may be an arbitrary partial order or may have a more restricted form, such as a tree or lattice. The naive algorithm for infering a most general polymorphic type, undervariable subtype hypotheses, requires deterministic exponential time. If we fix the subtype ordering, this upper bound grows to nondeterministic exponential time. We show that it is NP-hard to decide whether a lambda term has a type with respect to a fixed subtype hierarchy (involving only atomic type names). This lower bound applies to monomorphic or polymorphic languages. We give PSPACE upper bounds for deciding polymorphic typability if the subtype hierarchy has a lattice structure or the subtype hierarchy varies arbitrarily. We also give a polynomial time algorithm for the limited case where there are of no function constants and the type hierarchy is either variable or any fixed lattice.

[Liou1833a] Joseph Liouville. Premier mémoire sur la détermination des intégrales dont la valeur est algébrique. *Journal de l'Ecole Polytechnique*, 14:124–128, 1833.

[Liou1833b] Joseph Liouville. Second mémoire sur la détermination des intégrales dont la valeur est algébrique. *Journal de l'Ecole Polytechnique*, 14:149–193, 1833.

[Lips81] John D. Lipson. *Elements of Algebra and Algebraic Computing*. Addison-Wesley Educational Publishers, 1981, 978-0201041156. **Algebra:**
(p**??**) category FFIELDC FiniteFieldCategory

[Lisk77] Barbara Liskov, Alan Synder, Russell Atkinson, and Craig Schaffert. Abstraction Mechanisms in CLU. *CACM*, 20(8), 1977.

**Abstract:** CLU is a new programming language designed to support the use of abstractions in program construction. Work in programming methodology has led to the realization that three kinds of abstractions – procedural, control, and especially data abstractions – are useful in the programming process. Of these, only the procedural abstraction is supported well by conventional languages, through the procedure or subroutine. CLU provides, in addition to procedures, novel linguistic mechanisms that support the use of data and control abstractions. This paper provides an introduction to the abstraction mechanisms of CLU. By means of programming examples, the utility of the three kinds of abstractions in program construction is illustrated, and it is shown how CLU programs may be written to use and implement abstractions. The CLU library, which permits incremental program development with complete type checking performed at compile time, is also discussed.

**Link:** https://www.cs.virginia.edu/~weimer/615/reading/liskov-clu-abstraction.pdf

[Lisk79] Barbara Liskov, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler, and Alan Snyder. CLU Reference Manual. Technical report, Massachusetts Institute of Technology, 1979.

[Loos72] Rudiger Loos. Algebraic Algorithm Descriptions as Programs. *ACM SIGSAM Bulletin*, 23:16–24, 1972.

**Abstract:** We propose methods for writing algebraic programs in an algebraic notation. We discuss the advantages of this approach and a specific example

[Loos74] Ruediger G. K. Loos. Toward a Formal Implementation of Computer Algebra. *SIGSAM*, 8(3):9–16, 1974.

**Abstract:** We consider in this paper the task of synthesizing an algebraic system. Today the task is significantly simpler than in the pioneer days of symbol manipulation, mainly because of the work done by the pioneers in our area, but also because of the progress in other areas of Computer Science. There is now a considerable collection of algebraic algorithms at hand and a much better understanding of data structures and programming constructs than only a few years ago.

[Loos76] Rudiger Loos. The Algorithm Description Language (ALDES) (report). *ACM SIGSAM Bulletin*, 10(1):14–38, 1976.

**Abstract:** ALDES is a formalization of the method to describe algorithms used in Knuth's books. The largest documentation of algebraic algorithms, Collins' SAC system for Computer Algebra, is written in this language. In contrast to PASCAL it provides automatic storage deallocation. Compared to LISP equal emphasis was placed on efficiency of arithmetic, list processing, and array handling. To allow the programmer full control of efficiency all mechanisms of the system are accessible to him. Currently ALDES is available as a preprocessor to ANSI Fortran, using no additional primitives.

[Loos92] Rudiger Loos and George E. Collins. *Revised Report on the ALgorithm Language ALDES*. Institut fur Informatik, 1992.

[Loos93] Rudiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *The Computer Journal*, 36(5), 1993.

**Abstract:** The linear quantifier elimination algorithm for ordered fields in [15] is applicable to a wide range of examples involving even non-linear parameters. The Skolem sets of the original algorithm are generalized to elimination sets whose size is linear in the number of atomic formulas, whereas the size of Skolem sets is quadratic in this number. Elimiation sets may contain non-standard terms which enter the computation symoblically. Many cases can be treated by special methods improving further the empirical computing times.

[Lune87] H Lüneburg. On the Rational Normal Form of Endomorphisms, 1987.

**Comment:** BI-Wissenschaftsverlag

[Macl91] Saunders MacLane. *Categories for the Working Mathematician*. Springer, 1991,

0-387-98403-8.

**Link:** http://www.maths.ed.ac.uk/~aar/papers/maclanecat.pdf

[Macl92] Saunders MacLane. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Springer, 1992, 978-0-387-97710-2.

[Macq84] David MacQueen. Modules for Standard ML. In *Conf. on Lisp and Functional Programming*, pages 198–207. ACM Press, 1984.

**Abstract:** The functional programming language ML has been undergoing a thorough redesign during the past year, and the module facility described here has been proposed as part of the revised language, now called Standard ML. The design has three main goals: (1) to facilitate the structuring of large ML programs; (2) to support separate compilation and generic library units; and (3) to employ new ideas in the semantics of data types to extend the power of ML's polymorphic type system. It is based on concepts inherent in the structure of ML, primarily the notions of a declaration, its type signature, and the environment that it denotes.

[Macq86] David MacQueen. Using Dependent Types to Express Modular Structure. In *Principles of Programming Languages POPL'13*, pages 277–286. ACM Press, 1986.

**Link:** https://people.mpi-sws.org/~dreyer/courses/modules/macqueen86.pdf

[Macq88] David MacQueen. An Implementation of Standard ML Modules. In *Lisp and Functional Programming '88*, pages 212–223. ACM Press, 1988.

**Abstract:** Standard ML includes a set of module constructs that support programming in the large. These constructs extend ML's basic polymorphic type system by introducing the dependent types of Martin L/Sf's Intuitionistic Type Theory. This paper discusses the problems involved in implementing Standard ML's modules and describes a practical, efficient solution to these problems. The representations and algorithms of this implementation were inspired by a detailed formal semantics of Standard ML developed by Milner, Tofte, and Harper. The implementation is part of a new Standard ML compiler that is written ia Standard ML using the module system.

[Macq94] David MacQueen and Mads Tofte. A Semantics for Higher-order Functors. *LNCS*, 788:409–423, 1994.

**Abstract:** Standard ML has a module system that allows one to define parametric modules, called functors. Functors are "first-order," meaning that functors themselves cannot be passed as parameters or returned as results of functor applications. This paper presents a semantics for a higher-order module system which generalizes the module system of Standard ML. The higher-order functors described here are implemented in the current version of Standard ML of New Jersey and have proved useful in programming practice.

[Mane76] Ernest G. Manes. *Algebraic Theories*. Graduate Texts in Mathematics. Springer, 1976, 978-1-9860-1.

[Marc77] Daniel A. Marcus. *Number Fields*. Springer, 1977, 978-0387902791.

[Mart73] P. Martin-Lof. An Intuitionistic Theory of Types. In *Logic Colloquium 1973*, pages 73–118. H.E. Rose and J.G. Shepherdson, 1973.

**Abstract:** The theory of types with which we shall be concerned is intended to be a full scale system for foralizing intuitionistic mathematics as developed, for example, in the book by Bishop 1967. The language of the theory is richer than the language of first order predicate logic. This makes it possible to strengthen the axioms for existence and disjunction. In the case of existence, the possibility of strengthening the usual elimination rule seems first to have been indicated by Howard 1969, whose proposed axioms are special cases of the existential elimination rule of the present theory. Furthermore, there is a reflection principle which links the generation of objects and types and plays somewhat the same role for the present theory as does the replacement axiom for Zermelo-Frankel set theory. An earlier, not yet conclusive, attempt at formulating a theory of this kind was made by Scott 1970. Also related, although less closely, are the type and logic free theories of consructions of Kreisel 1962 and 1965 and Goodman 1970. In its first version, the present theory was based on the strongly impredicative axiom that there is a type of all types whatsoever, which is at the same time a type and an object of that type. This axiom had to be abandoned, however, after it was shown to lead to a contradiction by Jean Yves Girard. I am very grateful to him for showing me his paradox. The change that it necessitated is so drastic that my theory no longer contains intuitionistic simple type theory as it originally did. Instead, its proof theoretic strength should be close to that of predicative analysis.

[Mart79] P. Martin-Lof. Constructive Mathematics and Computer Programming. In *Proc. 6th Int. Congress for Logic, Methodology and Philosophy of Science*, pages 153–179. North-Holland, 1979.

**Abstract:** If programming is understood not as the writing of instructions for this or that computing machine but as the design of methods of computation that it is the computer's duty to execute (a difference that Dijkstra has referred to as the difference between computer science and computing science), then it no longer seems possible to distinguish the discipline of programming from constructive mathematics. This explains why the intuitionistic theory of types, which was originally developed as a symbolism for the precise codification of constructive mathematics, may equally well be viewed as a programming language. As such it provides a precise notation not only, like other programming languages, for the programs themselves but also for the tasks that the programs are supposed to perform. Moreover, the inference rules of the theory of types, which are again completely formal, appear as rules of correct program synthesis. Thus the correctness of a program written in the theory of types is proved formally at the same time as it is being synthesized.

[Mart80] Per Martin-Löf. Intuitionistic Type Theory, 1980.

**Link:** http://archive-pml.github.io/martin-lof/pdfs/Bibliopolis-Book-retypeset-1984.pdf

[Mart97] Ursula Martin and D Shand. Investigating some Embedded Verification Techniques for Computer Algebra Systems.

**Abstract:** This paper reports some preliminary ideas on a collaborative project between St. Andrews University in the UK and NAG Ltd. The project aims to use embedded verification techniques to improve the reliability and mathematical soundness of computer algebra systems. We give some history of attempts to integrate computer algebra systems and automated theorem provers and discuss possible advantages and disadvantages of these approaches. We also discuss some possible case studies.

**Link:** http://www.risc.jku.at/conferences/Theorema/papers/shand.ps.gz

[Mcca02] Scott McCallum and George E. Collins. Local box adjacency algorithms for cylindrical algebraic decompositions. *J. of Symbolic Computation*, 33(3):321–342, 2002.

**Abstract:** We describe new algorithms for determining the adjacencies between zero-dimensional cells and those one-dimensional cells that are sections (not sectors) in cylindrical algebraic decompositions (cad). Such adjacencies constitute a basis for determining all other cell adjacencies. Our new algorithms are local, being applicable to a specified 0D cell and the 1D cells described by specified polynomials. Particularly efficient algorithms are given for the 0D cells in spaces of dimensions two, three and four. Then an algorithm is given for a space of arbitrary dimension. This algorithm may on occasion report failure, but it can then be repeated with a modified isolating interval and a likelihood of success.

[Mcca84] Scott McCallum. *An Improved Projection Operator for Cylindrical Algebraic Decomposition*. PhD thesis, University of Wisconsin-Madison, 1984.

**Abstract:** A fundamental algorithm pertaining to the solution of polynomial equations in several variables is the *cylindrical algebraic decomposition (cad)* algorithm due to G.E. Collins. Given as input a set $A$ of integral polynomials in $r$ variables, the cad algorithm produces a decomposition of the euclidean space of $r$ dimensions into cells, such that each polynomial in $A$ is invariant in sign throughout each of the cells in the decomposition. A key component of the cad algorithm is the projection operation: the *projection* of a set $A$ of $r$-variate polynomials is defined to be a certain set $P$ of $(r-1)$-variate polynomials. The solution set, or variety, of the polynomials in $P$ comprises a projection in the geometric sense of the variety of $A$. The cad algorithm proceeds by forming successive projections of the input set $A$, each projection resulting in the elimination of one variable. This thesis is concerned with a refinement to the cad algorithm, and to its projection operation in particular. It is shown, using a theorem from real algebraic geometry, that the original projection set that Collins used can be substantially reduced in size, without affecting its essential properties. The results of theoretical analysis and empirical observations suggest that the reduction in the projection set size leads to an overall decrease in the computing time of the cad

algorithm.

**Comment:** Computer Sciences Technical Report #578

**Link:** ftp://ftp.cs.wisc.edu/pub/techreports/1985/TR578.pdf

[Mcca88] Scott McCallum. An improved projection operation for cylindrical algebraic decomposition of three-dimensional space. *J. Symbolic Computation*, 5(1-2):141–161, 1998.

**Abstract:** A key component of the cylindrical algebraic decomposition (cad) algorithm of Collins (1975) is the projection operation: the *projection* of a set $A$ of $r$-variate polynomials is defined to be a certain set or $(r-1)$-variate polynomials. Tile zeros of the polynomials in the projection comprise a "shadow" of the critical zeros of $A$. The cad algorithm proceeds by forming successive projections of the input set $A$, each projection resulting in the elimination of one variable. This paper is concerned with a refinement to the cad algorithm, and to its projection operation in particular. It is shown, using a theorem from complex analytic geometry, that the original projection set for trivariate polynomials that Collins used can be substantially reduced in size, without affecting its essential properties. Observations suggest that the reduction in the projection set size leads to a substantial decrease in the computing time of the cad algorithm.

[Mcca93] Scott McCallum. Solving Polynomial Strict Inequalities Using Cylindrical Algebraic Decomposition. *The Computer Journal*, 36(5):432–438, 1993.

**Abstract:** We consider the problem of determining the consistency over the real number of a system of integral polynomial strict inequalities. This problem has applications in geometric modelling. The cylindrical algebraic decomposition (cad) algorithm [2] can be used to solve this problem, though not very efficiently. In this paper we present a less powerful version of the cad algorithm which can be used to solve the consistency problem for conjunctions of strict inequalities, and which runs considerably faster than the original method applied to this problem. In the case that a given conjunction of strict inequalities is consistent, the modified cad algorithm constructs solution points with rational coordinates.

[Mend87] Elliot Mendelson. *Introduction to Mathematical Logic*. Wadsworth and Brooks/Cole, 1987, 978-1482237726.

[Meye86] Albert R. Meyer and Mark B. Reinhold. Type is not a type. In *POPL 86*, pages 287–295, 1986.

**Abstract:** A function has a dependent type when the type of its result depends upon the value of its argument. Dependent types originated in the type theory of intuitionistic mathematics and have reappeared independently in programming languages such as CLU, Pebble, and Russell. Some of these languages make the assumption that there exists a type-of-all-types which is its own type as well as the type of all other types. Girard proved that this approach is inconsistent from the perspective of intuitionistic logic. We apply Girard's techniques to establish that the type-of-all-types assumption creates

serious pathologies from a programming perspective: a system using this assumption is inherently not normalizing, term equality is undecidable, and the resulting theory fails to be a conservative extension of the theory of the underlying base types. The failure of conservative extension means that classical reasoning about programs in such a system is not sound.

[Meye88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

**Link:** https://sophia.javeriana.edu.co/~cbustaca/docencia/POO-2016-01/documentos/Object

[Meye92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992, 0-13-247925-7.

[Miln78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

**Abstract:** The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types, entails defining procedures which work well on objects of a wide variety. We present a formal type discipline for such polymorphic procedures in the context of a simple programming language, and a compile time type-checking algorithm $W$ which enforces the discipline. A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot go wrong and a Syntactic Soundness Theorem states that if $W$ accepts a program then it is well typed. We also discuss extending these results to richer languages; a type-checking algorithm based on $W$ is in fact already implemented and working, for the metalanguage ML in the Edinburgh LCF system,

**Link:** https://courses.engr.illinois.edu/cs421/sp2013/project/milner-polymorphism.pdf

[Miln90] Robin Milner, Mads Torte, and Robert Harper. *The Definition of Standard ML*. Lab for Foundations of Computer Science, Univ. Edinburgh, 1990.

**Link:** http://sml-family.org/sml90-defn.pdf

[Miln91] Robin Milner and Mads Torte. *Commentary on Standard ML*. Lab for Foundations of Computer Science, Univ. Edinburgh, 1991.

**Link:** https://pdfs.semanticscholar.org/d199/16cbbda01c06b6eafa0756416e8b6f15ff44.pdf

[Mish93] Bhubaneswar Mishra. *Algorithmic Algebra*. Texts and Monographs in Computer Sciences. Springer-Verlag, 1993.

**Abstract:** This book is based on a graduate course in computer science taught in 1987. The following topics are covered: computational ideal theory, solving systems of polynomial equations, elimination theory, real algebra, as well as an introduction chapter and two chapters with the needed algebraic background. The book is self-contained and the proofs are given with many details. It is clear that this book is only an introduction to the topic and does not cover the many improvements that appeared in the last 7 years about for example the computation of Groebner basis, polynomial solving, multivariate resultants and algorithms in real algebra. Choices had to be made to keep the content of a reasonable size and the complexity issues are not considered. The choice of topics is excellent, there are many exercises

and examples. It is a very useful book.

[Mitc88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM TOPLAS*, 10(3):470–502, 1988.

**Abstract:** Abstract data type declarations appear in typed programming languages like Ada, Alphard, CLU and ML. This form of declaration binds a list of identifiers to a type with associated operations, a composite value we call a data algebra. We use a second-order typed lambda calculus SOL to show how data algebras may be given types, passed as parameters, and returned as results of function calls. In the process, we discuss the semantics of abstract data type declarations and review a connection between typed programming languages and constructive logic.

[Mitc88a] John C. Mitchell. Polymorphic Type Inference and Containment. *Information and Computation*, 76(2-3):211–249, 1988.

**Abstract:** Type expressions may be used to describe the functional behavior of untyped lambda terms. We present a general semantics of polymorphic type expressions over models of untyped lambda calculus and give complete rules for inferring types for terms. Some simplified typing theories are studied in more detail, and containments between types are investigated.

[Mitc91] John C. Mitchell. TYpe Inference with Simple Subtypes. *J. of Functional Programming*, 1(3):245–285, 1991.

**Abstract:** Subtyping appears in a variety of programming languages, in the form of the automatic coercion of integers to reals, Pascal subranges, and subtypes arising from class hierarchies in languages with inheritance. A general framework based on untyped lambda calculus provides a simple semantic model of subtyping and is used to demonstrate that an extension of Curry's type inference rules are semantically complete. An algorithm G for computing the most general typing associated with any given expression, and a restricted, optimized algorithm GA using only atomic subtyping hypotheses are developed. Both algorithms may be extended to insert type conversion functions at compile time or allow polymorphic function declarations as in ML.

[Mitc91a] John C. Mitchell. Type Systems for Programming Languages. In *Handbook of Theoretical Computer Science (Vol B.)*, pages 365–458. MIT Press, 1991, 0-444-88074-7.

[Mitc91b] John Mitchell, Sigurd Meidal, and Neel Madhav. An Extension of Standard ML Modules with Subtyping and Inheritance. In *POPL'91*, pages 270–278, 1991, 0-89791-419-8.

**Abstract:** We describe a general module language integrating abstract data types, specifications and object-oriented concepts. The framework is based on the Standard ML module system, with three main extensions: subtyping, a form of object derived from ML structures, and inheritence primitives. The language aims at supporting a range of programming styles, including mixtures of object-oriented programming and programs built around specified algebraic or higher-

order abstract data types. We separate specification from implementation, and provide independent inheritance mechanisms for each. In order to support binary operations on objects within this framework, we introduce "internal interfaces// which govern the way that function components of one structure may access components of another. The language design has been tested by writing a number of program examples; an implementation is under development in the context of a larger project.

[Mona93] Michael B. Monagan. Gauss: a parameterized domain of computation system with support for signature functions. In *Design and Implementation of Symbolic Computation Systems*, Lecture Notes in Computer Science 722, pages 81–94, 1993, 3-540-57235-X.

**Abstract:** The fastest known algorithms in classical algebra make use of signature functions. That is, reducing computation with formulae to computing with the integers module $p$, by substituting random numbers for variables, and mapping constants module $p$. This isea is exploited in specific algorithms in computer algebra systems, e.g. algorithms for polynomial greatest common divisors. It is also used as a heuristic to speed up other calculations. But none exploit it in a systematic manner. The goal of this work was twofold. First, to design an AXIOM like system in which these signature functions can be constructed automatically, hence better exploited, and secondly, to exploit them in new ways. In this paper we report on the design of such a system, Gauss.

**Link:** http://www.cecm.sfu.ca/~monaganm/papers/DISCO93.pdf

[Monk76] J. Donald Monk. *Mathematical Logic*. Springer, 1976, 978-1-4684-9452-5.

[Muld97] Thom Mulders. A Note on Subresultants and the Lazard/Rioboo/Trager Formula in Rational Function Integration. *Journal of Symbolic Computation*, 24(1):45–50, July 1997.

**Abstract:** An ambiguity in a formula of Lazard, Rioboo and Trager, connecting subresultants and rational function integration, is indicated and examples of incorrect interpretations are given.

[Mull88] R.C. Mullin, I.M. Onyszchuk, and S.A. Vanstone. Optimal Normal Bases in $GF(p^n)$. *Discrete Applied Mathematics*, 22:149–161, 1988.

[Nick88] W. Nickel. Endliche Körper in dem gruppentheoretischen Programmsystem GAP, 1988.

**Comment:** Diplomarbeit, RWTH Aachen

[Nipk91] Tobias Nipkow and Gregor Snelting. Type Classes and Overloading Resolution via Order-Sorted Unification. In *Proc 5th ACM Conf. Functional Prog. Lang. and Comp. Arch.*, volume 523, pages 1–14. Springer, 1991.

**Abstract:** We present a type inference algorithm for a Haskell-like language based on order-sorted unification. The language features polymorphism, overloading, type classes and multiple inheritance. Class and instance declarations give rise to an order-sorted algebra of types. Type inference essentially reduces to the Hindley/Milner algorithm where unification takes place in this order-sorted algebra of

types. The theory of order-sorted unification provides simple sufficient conditions which ensure the existence of principal types. The semantics of the language is given by a translation into ordinary lambda-calculus. We prove the correctness of our type inference algorithm with respect to this semantics.

[Nipk95] Tobias Nipkow and Christian Prehofer. Type Reconstruction for Type Classes. *J. of Functional Programming*, pages 201–224, 1995.

**Abstract:** We study the type inference problem for a system with type classes as in the functional programming language Haskell. Type classes are an extension of ML-style polymorphism with overloading. We generalize Milner's work on polymorphism by introducing a separate context constraining the type variables in a typing judgement. This leads to simple type inference systems and algorithms which closely resemble those for ML. In particular we present a new unification algorithm which is an extension of syntactic unification with constraint solving. The existence of principal types follows from an analysis of this unification algorithm.

[Odif92] Piergiorgio Odifreddi. *Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers*. Elsevier, 1992.

[Odly85] A.M. Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. *Proc. Eurocrypt '84, LNCS*, 209:224–314, 1985.

[Ostr1845] Ostrogradsky. M.W. De l'intégration des fractions rationelles., 1845.

[Pada80] Peter Padawitz. New results on completeness and consistency of abstract data types. *LNCS*, 88:460–473, 1980.

**Abstract:** If an algebraic specification is designed in a structured way, a small specification is stepwise enriched by more complex operations and their defining equations. Based on normalization properties of term reductions we present sufficient "local" conditions for the completeness and consistency of enrichment steps, which can be efficiently verified in many cases where other attempts to prove the enrichment property "syntactically" have failed so far.

[Pate78] M. S. Paterson. Linear Unification. *J. Computer and System Sciences*, 16(2):158–167, 1978.

**Abstract:** A unification algorithm is described which tests a set of expressions for unifiability and which requires time and space which are only linear in the size of the input

[Pede93] P. Pedersen, F.-F. Roy, and A. Szpirglas. Counting real zeroes in the multivariate case. In *Proc. MEGA'92: Computational Algebraic Geometry*, volume 109, pages 203–224, 1993.

**Abstract:** In this paper we show, by generalizing Hermites theorem to the multivariate setting, how to count the number of real or complex points of a discrete algebraic set which lie within some algebraic constraint region. We introduce a family of quadratic forms determined by the algebraic constraints and defined in terms of the trace from the coordinate ring of the variety to the ground field, and we show that the rank and signature of these forms are sufficient to de-

termine the number of real points lying within a constraint region. In all cases we count geometric points, which is to say, we count points without multiplicity. The theoretical results on these quadratic forms are more or less classical, but forgotten too, and can be found also in [3]. We insist on effectivity of the computation and complexity analysis: we show how to calculate the trace and signature using Grbner bases, and we show how the information provided by the individual quadratic forms may be combined to determine the number of real points satisfying a conjunction of constraints. The complexity of the computation is polynomial in the dimension as a vector space of the quotient ring associated to the defining equations. In terms of the number of variables, the complexity of the computation is singly exponential. The algorithm is well parallelizable. We conclude the paper by applying our machinery to the problem of effectively calculating the Euler characteristic of a smooth hypersurface.

[Pfen89] Frank Pfenning and Christine Paulin-Mohring. Inductively Defined Types in the Calculus of Constructions. Technical Report CMU-CS-89-209, Carnegie-Mellon University, 1989.

**Abstract:** We define the notion of an *inductively defined type* in the Calculus of Constructions and show how inductively defined types can be represented by closed types. We show that all primitive recursive functional over these inductively defined types are also representable. This generalizes work by Bohm and Berarducci on synthesis of functions on term algebras in the second-order polymorphic $\lambda$-calculus ($F_2$). We give several applications of this generalization, including a representation of $F_2$-programs in $F_3$, along with a definition of functions **reify**, **reflect**, and **eval** for $F_2$ in $F_3$. We also show how to define induction over inductively defined types and sketch some results that show that the extension of the Calculus of Construction by induction principles does not alter the set of functions in its computational fragment, $F_\omega$. This is because a proof by induction can be **realized** by primitive recursion, which is already definable in $F_\omega$.

**Link:** http://repository.cmu.edu/cgi/viewcontent.cgi?article=2907&context=compsci

[Pfen91] Frank Pfenning. Logic Programming in the LF Logical Framework. In *Proc. First Workshop on Logical Frameworks*, 1991.

[Pfen91a] Frank Pfenning. Unification and Anti-Unification in the Calculus of Constructions. In *Logic in Computer Science 91*, pages 74–85, 1991.

**Abstract:** We present algorithms for unification and anti-unification in the Calculus of Constructions, where occurrences of free variables (the variables subject to instantiation) are restricted to higher-order patterns, a notion investigated for the simply-typed $\lambda$-calculus by Miller. Most general unifiers and least common anti-instances are shown to exist and are unique up to a simple equivalence. The unification algorithm is used for logic program execution and type and term reconstruction in the current implementation of Elf and has shown itself to be practical. The main application of the anti-unification algorithm we have in mind is that of proof generalization.

[Pfen92] Frank Pfenning. *Types in Logic Programming.* MIT Press, 1992, 9780262161312.

**Abstract:** Types play an increasingly important role in logic programming, in language design as well as language implementation. We present various views of types, their connection, and their role within the logic programming paradigm. Among the basic views of types we find the so-called descriptive systems, where types describe properties of untyped logic programs, and prescriptive systems, where types are essential to the meaning of programs. A typical application of descriptive types is the approximation of the meaning of a logic program as a subset of the Herbrand universe on which a predicate might be true. The value of prescriptive systems lies primarily in program development, for example, through early detection of errors in programs which manifest themselves as type inconsistencies, or as added documentation for the intended and legal use of predicates. Central topics within these views are the problems of type inference and type reconstruction, respectively. Type inference is a form of analysis of untyped logic programs, while type reconstruction attempts to fill in some omitted type information in typed logic programs and generalizes the problem of type checking. Even though analogous problems arise in functional programming, algorithms addressing these problems are quite different in our setting. Among the specific forms of types we discuss are simple types, recursive types, polymorphic types, and dependent types. We also briefly touch upon subtypes and inheritance, and the role of types in module systems for logic programming languages.

[Pier91] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, Carnegie Mellon University, 1991.

**Abstract:** Intersection types and bounded quantification are complementary mechanisms for extending the expressive power of statically typed programming languages. They begin with a common framework: a simple, typed language with higher-order functions and a notion of subtyping. Intersection types extend this framework by giving every pair of types $\sigma$ and $\tau$ a greatest lower bound, $\sigma \wedge \tau$, corresponding intuitively to the intersection of the sets of values described by $\sigma$ and $\tau$. Bounded quantification extends the basic framework along a different axis by adding polymorphic functions that operate uniformly on all the subtypes of a given type. This thesis unifies and extends prior work on intersection types and bounded quantification, previously studied only in isolation, by investigating theoretical and practical aspects of a typed $\lambda$-calculus incorporating both. The practical utility of this calculus, called $F_\wedge$ is established by examples showing, for instance, that it allows a rich form of "coherent overloading" and supports an analog of abstract interpretation during typechecking; for example, the addition function is given a type showing that it maps pairs of positive inputs to a positive result, pairs of zero inputs to a zero result, etc. More familiar programming examples are presented in terms of an extention of Forsythe (an Algol-like language with intersection types), demonstrating how parametric polymorphism can

be used to simplify and generalize Forsythe's design. We discuss the novel programming and debugging styles that arise in $F_\wedge$. We prove the correctness of a simple semi-decision procedure for the subtype relation and the partial correctness of an algorithm for synthesizing minimal types of $F_\wedge$ terms. Our main tool in this analysis is a notion of "canonical types," which allows proofs to be factored so that intersections are handled separately from the other type constructors. A pair of negative results illustrates some subtle complexities of $F_\wedge$. First, the subtype relation of $F_\wedge$ is shown to be undecidable; in fact, even the sutype relation of pure second-order bounded quantification is undecidable, a surprising result in its own right. Second, the failure of an important technical property of the subtype relation – the existence of least upper bounds – indicates that typed semantic models of $F_\wedge$ will be more difficult to construct and analyze than the known typed models of intersection types. We propose, for future study, some simpler fragments of $F_\wedge$ that share most of its essential features, while recovering decidability and least upper bounds. We study the semantics of $F_\wedge$ from several points of view. An untyped model based on partial equivalence relations demonstrates the consistency of the typing rules and provides a simple interpolation for programs, where "$\sigma$ is a subtype of $\tau$" is read as "$\sigma$ is a subset of $\tau$." More refined models can be obtained using a translation from $F_\wedge$ into the pure polymorphic $\lambda$-calculus; in these models, "$\sigma$ is a subtype of $\tau$" is interpreted by an explicit coercion function from $\sigma$ to $\tau$. The nonexistence of least upper bounds shows up here in the failure of known techniques for proving the coherence of the translation semantics. Finally, an equational theory of equivalences between $F_\wedge$ terms is presented and its soundness for both styles of model is verified.

**Comment:** CMU-CS-91-205

[Pier91a] Benjamin C. Pierce. Bounded Quantification is Undecidable. Technical Report CMU-CS-91-161, Carnegie Mellon University, 1991.

**Abstract:** $F_\leq$ is a typed $\lambda$-calculus with subtyping and bounded second-order polymorphism. First introduced by Cardelli and Wegner, it has been widely studied as a core calculus for type systems with subtyping. Curien and Ghelli proved the partial correctness of a recursive procedure for computing minimal types of $F_\leq$ terms and showed that the termination of this procedure is equivalent to the termination of its major component, a procedure for checking the subtype relation between $F_\leq$ types. Ghelli later claimed that this procedure is also guaranteed to terminate, but the discovery of a subtle bug in his proof led him recently to observe that, in fact, there are inputs on which the subtyping procedure diverges. This reopens the question of the decidability of subtyping and hence of typechecking. This question is settled here in the negative, using a reduction from the halting problem for two-counter Turing machines to show that the subtype relation of $F_\leq$ is undecidable.

**Link:** http://repository.cmu.edu/cgi/viewcontent.cgi?article=3059

[Pier93] Benjamin C. Pierce and David N. Turner. Object-oriented Programming with-

out Recursive Types. In *POPL'93*, pages 299–312, 1993.

**Abstract:** It is widely agreed that recursive types are inherent in the static typing of the essential mechanisms of object-oriented programming: encapsulation, message passing, subtyping, and inheritance. We demonstrate here that modeling object encapsulation in terms of existential types yields a substantially more straightforward explanation of these features in a simpler calculus without recursive types.

[Pinc89] A Pincin. Bases for finite fields and a canonical decomposition for a normal basis generator. *Communications in Algebra*, 17(6):1337–1352, 1989.

[Pohl78] S.C. Pohlig and M. Hellman. An improved algorith for computing logarithms over $GF(p)$ and its cryptographic significance. *IEEE Trans Information Theory*, IT-24:106–110, 1978.

[Poiz85] B. Poizat. Cours de Théorie des Modéles, 1985.

**Comment:** Nur al-Mantiq wal-Ma'rifah, Villeurbanne, France

[Pome80] C. Pomerance, J.L. Slefridge, and S.S. Wagstaff Jr. The Pseudoprimes up to $25 \cdot 10^9$. *Math. Comp.*, 35:1003–1026, 1980.

[Puff09] Puffinware LLC. Singular Value Decomposition (SVD) Tutorial.

**Link:** http://www.puffinwarellc.com/p3a.htm

[Rabi80] M.O. Rabin. Probabilistic Algorithm for Testing Primality. *J. Number Theory*, 12:128–138, 1980.

[Rats02] Stefan Ratschan. Approximate quantified constraint solving by cylindrical box decomposition. *Reliable Computing*, 8(1):21–42, 2002.

**Abstract:** This paper applies interval methods to a classical problem in computer algebra. Let a quantified constraint be a first-order formula over the real numbers. As shown by A. Tarski in the 1930's, such constraints, when restricted to the predicate symbols $<$, $=$ and function symbols $+$, $\times$, are in general solvable. However, the problem becomes undecidable, when we add function symbols like sin. Furthermore, all exact algorithms known up to now are too slow for big examples, do not provide partial information before computing the total result, cannot satisfactorily deal with interval constants in the input, and often generate huge output. As a remedy we propose an approximation method based on interval arithmetic. It uses a generalization of the notion of cylindrical decompositionas introduced by G. Collins. We describe an implementation of the method and demonstrate that, for quantified constraints without equalities, it can efficiently give approximate information on problems that are too hard for current exact methods.

[Rect89] D. L. Rector. Semantics in Algebraic Computation. In *Computers and Mathematics*, pages 299–307. Springer-Verlag, 1989, 0-387-97019-3.

**Abstract:** I am interested in symbolic computation for theoretical research in algebraic topology. Most algebraic computations in topology are hand calculations; that is, they can be accomplished by the researcher in times ranging from hours to weeks, and they are aimed at discovering general patterns rather than producing specific formulas

understood in advance. Furthermore, the range of algebraic constucts used in such calculations is very wide.

[Remy89] Didier Remy. Typechecking Records and Variants in a Natural Extension of ML. In *POPL 89*. ACM, 1989, 978-0-89791-294-5.

**Abstract:** We describe an extension of ML with records where inheritance is given by ML generic polymorphism. All common operations on records but concatenation are supported, in particular, the free extension of records. Other operations such as renaming of fields are added. The solution relies on an extension of ML, where the language of types is sorted and considered modulo equations, and on a record extension of types. The solution is simple and modular and the type inference algorithm is efficient in practice.

**Link:** https://www.cs.cmu.edu/~aldrich/courses/819/row.pdf

[Rene92] James Renegar. On the computational complexity and geometry of the first-order theory of the reals. Part I: Introduction. *J. of Symbolic Computation*, 13(3):255–299, 1992.

**Abstract:** This series of papers presents a complete development and complexity analysis of a decision method, and a quantifier elimination method, for the first order theory of the reals. The complexity upper bounds which are established are the best presently available, both for sequential and parallel computation, and both for the bit model of computation and the real number model of computation; except for the bounds pertaining to the sequential decision method in the bit model of computation, all bounds represent significant improvements over previously established bounds.

**Link:** http://www.sciencedirect.com/science/article/pii/S0747717110800033

[Rene92a] James Renegar. On the computational complexity and geometry of the first-order theory of the reals. Part II: The general decision problem. *J. of Symbolic Computation*, 13(3):301–327, 1992.

**Abstract:** This series of papers presents a complete development and complexity analysis of a decision method, and a quantifier elimination method, for the first order theory of the reals. The complexity upper bounds which are established are the best presently available, both for sequential and parallel computation, and both for the bit model of computation and the real number model of computation; except for the bounds pertaining to the sequential decision method in the bit model of computation, all bounds represent significant improvements over previously established bounds.

**Link:** http://www.sciencedirect.com/science/article/pii/S0747717110800045

[Rene92b] James Renegar. On the computational complexity and geometry of the first-order theory of the reals. Part III: Quantifier elimination. *J. of Symbolic Computation*, 13(3):329–352, 1992.

**Abstract:** This series of papers presents a complete development and complexity analysis of a decision method, and a quantifier elimination method, for the first order theory of the reals. The complexity upper bounds which are established are the best presently available, both

for sequential and parallel computation, and both for the bit model of computation and the real number model of computation; except for the bounds pertaining to the sequential decision method in the bit model of computation, all bounds represent significant improvements over previously established bounds.

**Link:** http://www.sciencedirect.com/science/article/pii/S0747717110800057

[Reyn74] John C. Reynolds. Towards a Theory of Type Structure. In *Colloquim on Programming*, pages 408–425, 1974.

[Reyn80] John C. Reynolds. Using Category Theory to Design Implicit Conversions and Generic Operators. In *Lecture Notes in Computer Science*, 1980.

**Abstract:** A generalization of many-sorted algebras, called category-sorted algebras, is defined and applied to the language-design problem of avoiding anomalies in the interaction of implicit conversions and generic operators. The definition of a simple imperative language (without any binding mechanisms) is used as an example.

[Reyn84] John C. Reynolds. Polymorphism is not Set-theoretic. In *Proc Semantics of Data Types*, pages 145–156, 1984.

**Abstract:** The polymorphic, or second-order, typed lambda calculus is an extension of the typed lambda calculus in which polymorphic functions can be defined. In this paper that the standard set-theoretic model of the ordinary typed lambda calculus cannot be extended to model this language extension.

**Link:** https://hal.inria.fr/inria-00076261/document

[Reyn91] John C. Reynolds. The Coherence of Languages with Intersection Types. In *TACS 91*, 1991.

**Abstract:** When a programming language has a sufficiently rich type structure, there can be more than one proof of the same typing judgement; potentially this can lead to semantic ambiguity since the semantics of a typed language is a function of such proofs. When no such ambiguity arises, we say that the language is coherent. In this paper we prove the coherence of a class of lambda-calculus-based languages that use the intersection type discipline, including both a purely functional programming language and the Algol-like programming language Forsythe.

[Risc68] Robert Risch. On the integration of elementary functions which are built up using algebraic operations. Research Report SP-2801/002/00, System Development Corporation, Santa Monica, CA, USA, 1968.

**Abstract:** This paper advances the study of the problem of integration of elementary functions in finite terms to within one step of a complete solution. A previous paper gave an algorithm for integrating those elementary functions which are built up using rational operations, exponentials and logarithms, under the condition that the exponentials and logarithms could not be replaced by adjoining constants and performing algebraic operations. Now it is show that with algebraic operations allowed, the problem reduces to a problem in the theory of algebraci functions which is believed to be decidable.

[Risc69a] Robert Risch. Further results on elementary functions. Research Report RC-2042, IBM Research, Yorktown Heights, NY, USA, 1969.

[Risc69b] Robert Risch. The problem of integration in finite terms. *Transactions of the American Mathematical Society*, 139:167–189, 1969.

> **Abstract:** This paper deals with the problem of telling whether a given elementary function, in the sense of analysis, has an elementary indefinite integral.

[Risc70] Robert Risch. The Solution of the Problem of Integration in Finite Terms. *Bull. AMS*, 76(3):605–609, 1970.

> **Abstract:** The problem of integration in finite terms asks for an algorithm for deciding whether an elementary function has an elementary indefinite integral and for finding the integral if it does. "Elementary" is used here to denote those functions build up from the rational functions using only exponentiation, logarithms, trigonometric, inverse trigonometric and algebraic operations. This vaguely worded question has several precise, but inequivalent formulations. The writer has devised an algorithm which solves the classical problem of Liouville. A complete account is planned for a future publication. The present note is intended to indiciate some of the ideas and techniques involved.

[Risc79] Robert H. Risch. Algebraic Properties of the Elementary Functions of Analysis. *American Journal of Mathematics*, 101(4):743–759, 1979.

> **Abstract:** The elementary functions of a complex variable $z$ are those functions built up from the rational functions of $z$ by exponentiation, taking logarithms, and algebraic operations. The purpose of this paper is first, to prove a 'structure theorem' which shows that if an algebraic relation holds among a set of elementary functions, then they must satisfy an algebraic relation of a special kind. Then we make four applications of this theorem, obtaining both new and old results which are described here briefly (and imprecisely).
>
> 1. An algorithm is given for telling when two elementary expressions define the same function.
> 2. A characterization is derived of those ordinary differential equations having elementary solutions
> 3. The four basic functions of elementary calculus – exp, log, tan, $\tan^{-1}$, – are shown to be 'irredundant'
> 4. A characterization is given of elementary functions possessing elementary inverses.

[Robi96] J. S. Derek Robinson. *A Course in the Theory of Groups*. Graduate Texts in Mathematics. Springer, 1996, 978-1-4612-6443-9.

[Roll1691] Michel Rolle. Rolle's Therem, 1691.

> **Abstract:** If a real-valued function $f$ is continuous on a proper closed interval $[a, b]$, differentiable on the open interval $(a, b)$, and $f(a) = f(b)$, then there exists at least one $c$ in the open interval $(a, b)$ such that $f'(c) = 0$.

**Link:** https://en.wikipedia.org/wiki/Rolle%27s_theorem

[Rose72] Maxwell Rosenlicht. Integration in finite terms. *American Mathematical Monthly*, 79:963–972, 1972.

[Roth77] Michael Rothstein. A new algorithm for the integration of exponential and logarithmic functions. *Proceedings of the 1977 MACSYMA Users Conference*, pages 263–274, 1977.

[Rybo89] M Rybowicz. Search of primitive polynomials over finite fields. *J. Pure Appl.*, 65:139–151, 1989.

[Ryde88] D. E. Rydeheard and R. M. Burstall. *Computational Category Theory*. Prentice Hall, 1988, 978-0131627369.

[Salo16] Matthew Salomone. Exploring Abstract Algebra II, 2016.

**Link:** https://www.youtube.com/watch?v=RNpdUG_yH_s&list=PLL0ATV5XYF8DTGAPKRPtYa4E8rOLcw

[Sant95] Philip S. Santas. A Type System for Computer Algebra. *J. Symbolic Computation*, 19(1-3):79–109, 1995.

**Abstract:** This paper presents a type system for support of subtypes, parameterized types with sharing and categories in a computer algebra environment. By modeling representation of instances in terms of existential types, we obtain a simplified model, and build a basis for defining subtyping among algebraic domains. The inheritance at category level has been formalized; this allows the automatic inference of type classes. By means of type classes and existential types we construct subtype relations without involving coercions. A type sharing mechanism works in parallel and allows the consistent extension and combination of domains. The expressiveness of the system is further increased by viewing domain types as special case of package types, forming weak and strong sums respectively. The introduced system, although awkward at first sight, is simpler than other proposed systems for computer algebra without including some of their problems. The system can be further extended in other to support more constructs and increase its flexibility.

[Sche93] Alfred Scheerhorn. Presentation of the algebraic closure of finite fields and trace-compatible polynomial sequences, 1993.

**Abstract:** For numerical experiments concerning various problems in a finite field $\mathbb{F}_q$ it is useful to have an explicit data presentation $\mathbb{F}_{q^m}$ of for large $m$, and a method for the construction of towers

$$\mathbb{F}_q \subset \mathbb{F}_{q^{d_1}} \subset \cdots \subset \mathbb{F}_{q^{d_k}} = \mathbb{F}_{q^m}$$

In order to avoid the identification problem it is advantageous to have all fields in the tower presented by properly chosen normal bases, whereby the embedding $\mathbb{F}_{q^{d_i}} \subset \mathbb{F}_{q^{d_{i+1}}}$ is given by the trace function. The following notion is introduced: A sequence of polynomials $\{f_n | n \geq 1\}$ with $\text{degree}(f_n) = n$ called trace-compatible over $\mathbb{F}_q$ if (1) $f_n$ is a normal polynomial over $\mathbb{F}_q$, (2) if $\alpha_n \in \mathbb{F}_{q^n}$ is a root of $f_n$, then for any proper divisor $d$ of $n$ the trace of $\alpha_n$ over $\mathbb{F}_{q^d}$ is a root of $f_d$. The main goal of the dissertation is to give algorithms for construction of sequences of trace-compatible polynomials and to present explicit

numerical data. An analogous notion of norm-compatible sequences is also introduced and studied. The dissertation consists of four chapters and a supplement, as follows: (1) Basic notions (1-31). (2) Presentation of the algebraic closure of a finite field (32-59). (3) Sequences of polynomials and sequences of elements (60-115). (4) Implementations (118-139). (5) Supplement (142-171). In chapters (1)(3) various known results and algorithms are collected, and new results are added and compared with those previously used. The numerical results in the supplement contain sequences of trace-compatible polynomials of degree $n$, where $n \leq 100$, and $q = 2, 3, 5, 7, 11, 13$. For implementation, the computer-algebra system AXIOM has been used. The details contained in this dissertation are not readily describable in a short review.

**Comment:** Darstellungen des algebraischen Abschlusses endlicher Korper und spur-kompatible Polynomfolgen

[Schm89] M. Schmidt-Schauss. *Computational Aspects of an Order-Sorted Logic with Term Declarations*. Springer, 1989, 978-3-540-51705-4.

[Scho24] M. Schoenfinkel. Uber die Bausteine der mathematischen Logik, 1924.

[Schu72] Horst Schubert. *Categories*. Springer-Verlag, 1972.

[Schw85] Fritz Schwarz. An Algorithm for Determining Polynomial First Integrals of Autonomous Systems of Ordinary Differential Equations. *J. Symbolic Computation*, 1:229–233, 1985.

[Schw87] Fritz Schwarz. Programming with abstract data types: the symmetry package SPDE in Scratchpad'. In *Trends in Computer Algebra*, Lecture Notes in Computer Science 296, pages 167–176, 1987, 3-540-18928-9.

**Abstract:** The main problem which occurs in developing Computer Algebra packages for special areas in mathematics is the complexity. The unique concept which is advocated to cope with that problem is the introduction of suitable abstract data types. The corresponding decomposition into modules makes it much easier to develop, maintain and change the program. After introducing the relevant concepts from software engineering they are elaborated by means of the symmetry analysis of differential equations and the Scratchpad package SPDE which abbreviates Symmetries of Partial Differential Equations.

[Seid54] A. Seidenberg. A New Decision Method for Elementary Algebra. *Annals of Mathematics*, 60(2):365–374, 1954.

**Abstract:** A. Tarski [4] has given a decision method for elementary algebra. In essence this comes to giving an algorithm for deciding whether a given finite set of polynomial inequalities has a solution. Below we offer another proof of this result of Tarski. The main point of our proof is accomplished upon showing how to decide whether a given polynomial $f(x, y)$ in two variables, defined over the field $\mathbb{R}$ of rational numbers, has a zero in a real-closed field $\mathbb{K}$ containing $R^1$. This is done in §2, but for purposes of induction it is necessary to consider also the case that the coefficients of $f(x, y)$ involve parameters; the remarks in §3 will be found sufficient for this point. In §1, the problem is

reduced to a decision for equalities, but an induction (on the number of unknowns) could not possibly be carried out on equalities alone; we consider a simultaneous system consisting of one equality $f(x, y) = 0$ and one inequality $F(x) \neq 0$. Once the decision for this case is achieved, at least as in §3, the induction is immediate.

[Shou92] V. Shoup. Searching for Primitive Roots in Finite Fields. *Math. of Comp.*, 58(197):369–380, 1992.

[Siek89] Jorg H. Siekmann. Unification Theory. *Journal of Symbolic Computation*, 7(3-4):207–274, 1989.

**Abstract:** Most knowledge based systems in artificial intelligence (AI), with a commitment to asymbolic representation, support one basic operation: "matching of descriptions". This operation, called unification in work on deduction, is the "addition-and-multiplication" of AI-systems and is consequently often supported by special purpose hardware or by a fast instruction set on most AI-machines. Unification theory provides the formal framework for investigations into the properties of this operation. This article surveys what is presently known in unification theory and records its early history.

[Smol88] G. Smolka. Logic Programming with Polymorphically Order-sorted Types. *Lecture Notes in Computer Science*, 343:53–70, 1988.

[Smol89] G. Smolka, W. Nutt, J. Goguen, and J. Meseguer. Order-sorted Equational Computation. In *Resolution of Equations in Algebra Structures (Vol 2)*, pages 297–367. Academic Press, 1989.

[Smol89a] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, Fachbereich Informatik, Universitat Kaiserslautern, 1989.

[Stan88] R. Stansifer. Type Inference with Subtypes. In *POPL 88*, pages 88–97, 1988.

**Abstract:** We give an algorithm for type inference in a language with functions, records, and variant records. A similar language was studied by Cardelli who gave a type checking algorithm. This language is interesting because it captures aspects of object-oriented programming using subtype polymorphism. We give a type system for deriving types of expressions in the language and prove the type inference algorithm is sound, i.e., it returns a type derivable from the proof system. We also prove that the type the algorithm finds is a "principal" type, i.e., one which characterizes all others. The approach taken here is due to Milner for universal polymorphism. The result is a synthesis of subtype polymorphism and universal polymorphism.

[Stic93] H. Stichtenoth. Algebraic function fields and codes, 1993.

[Stin90] D.R. Stinson. Some observations on parallel Algorithms for fast exponentiation in $GF(2^n)$. *Siam J. Comp.*, 19(4):711–717, 1990.

**Abstract:** A normal basis represention in $GF(2^n)$ allows squaring to be accomplished by a cyclic shift. Algorithms for multiplication in $GF(2^n)$ using a normal basis have been studied by several researchers. In this paper, algorithms for performing exponentiation in $GF(2^n)$ using a normal basis, and how they can be speeded up by using parallelization, are investigated.

**Algebra:**
(p**??**) package INBFF InnerNormalBasisFieldFunctions

[Stra00] Christopher Strachey. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, 13(1-2):11–49, 2000.

**Abstract:** This paper forms the substance of a course of lectures given at the International Summer School in Computer Programming at Copenhagen in August, 1967. The lectures were originally given from notes and the paper was written after the course was finished. In spite of this, and only partly because of the shortage of time, the paper still retains many of the shortcomings of a lecture course. The chief of these are an uncertainty of aimit is never quite clear what sort of audience there will be for such lecturesand an associated switching from formal to informal modes of presentation which may well be less acceptable in print than it is natural in the lecture room. For these (and other) faults, I apologise to the reader. There are numerous references throughout the course to CPL [13]. This is a programming language which has been under development since 1962 at Cambridge and London and Oxford. It has served as a vehicle for research into both programming languages and the design of compilers. Partial implementations exist at Cambridge and London. The language is still evolving so that there is no definitive manual available yet. We hope to reach another resting point in its evolution quite soon and to produce a compiler and reference manuals for this version. The compiler will probably be written in such a way that it is relatively easyto transfer it to another machine, and in the first instance we hope to establish it on three or four machines more or less at the same time. The lack of a precise formulation for CPL should not cause much difficulty in this course, as we are primarily concerned with the ideas and concepts involved rather than with their precise representation in a programming language.

[Stro95] Bjarne Stroustrup. *The C++ Programming Language (2nd Edition)*. Addison-Wesley, 1995, 0-201-53992-6.

[Stur1829] Jacques Charles Francois Sturm. Mémoire sur la rśolution des équations numériques. *Bulletin des Sciences de Férussac*, 11:419–425, 1829.

**Abstract:** Let $p_0, \ldots, p_m$ be the Sturm chain of a square free polynomial $p$, and let $\sigma\eta$ denote the number of sign changes (ignoring zeros) in the sequence $p_0(\eta), p_1(\eta), p_2(\eta), \ldots, p_m(\eta)$. Sturms' theorem states that for two real numbers $a < b$, the number of distinct roots of $p$ in the half-open interval $(a, b]$ is $\sigma(a) - \sigma(b)$. A Sturm chain is a finite sequence of polynomials $p_0, p_1, \ldots, p_m$ of decreasing degree with these following properties:

- $p_0 = p$ is square free (no square factors, i.e. no repeated roots)
- if $p(\eta) = 0$, then $sign(p_1(\eta)) = sign(p'(\eta))$
- if $p_i(\eta) = 0$ for $0 < i < m$ then $sign(p_{i-1}(\eta)) = -sign(p_{i+1}(\eta))$
- $p_m$ does not change its sign

**Link:** https://en.wikipedia.org/wiki/Sturm%27s_theorem

[Suto87] Robert S. Sutor and Richard D. Jenks. The type inference and coercion facilities in the Scratchpad II interpreter. *SIGPLAN Notices*, 22(7):56–63, 1987, 0-89791-235-7.

**Abstract:** The Scratchpad II system is an abstract datatype programming language, a compiler for the language, a library of packages of polymorphic functions and parametrized abstract datatypes, and an interpreter that provides sophisticated type inference and coercion facilities. Although originally designed for the implementation of symbolic mathematical algorithms, Scratchpad II is a general purpose programming language . This paper discusses aspects of the implementation of the interpreter and how it attempts to provide a user friendly and relatively weakly typed front end for the strongly typed programming language.

**Comment:** IBM Research Report RC 12595 (#56575)

[Szab82] P. Szabo. *Unifikationstheorie erster Ordnung*. PhD thesis, Fakultät für Informatik, Universität Karlsruhe, 1982.

[Tait1890] P.G. Tait. *An Elementary Treatise on Quaternions*. C.J. Clay and Sons, Cambridge University Press Warehouse, Ave Maria Lane, 1890.

[Tars48] Alfred Tarski. A Decision Method for Elementary Algebra and Geometry, 1948.

**Link:** https://www.rand.org/content/dam/rand/pubs/reports/2008/R109.pdf

[Temp92] M. Temperini. Design and Implementation Methodologies for Symbolic Computation Systems, 1992.

**Comment:** Preprint

[That91] Satish R. Thatte. Coercive Type Isomorphism. *LNCS*, 523:29–49, 1991.

**Abstract:** There is a variety of situations in programming in which it is useful to think of two distinct types as representations of the same abstract structure. However, language features which allow such relations to be effectively expressed at an abstract level are lacking. We propose a generalization of ML-style type inference to deal effectively with this problem. Under the generalization, the (normally free) algebra of type expressions is subjected to an equational theory generated by a finite set of user-specified equations that express interconvertibility relations between objects of "equivalent" types. Each type equation is accompanied by a pair of conversion functions that are (at least partial) inverses. We show that so long as the equational theory satisfies a reasonably permissive syntactic constraint, the resulting type system admits a complete type inference algorithm that produces unique principal types. The main innovation required in type inference is the replacement of ordinary free unification by unification in the user-specified equational theory. The syntactic constraint ensures that the latter is unitary, i.e., yields unique most general unifiers. The proposed constraint is of independent interest as the first known syntactic characterization for a class of unitary theories. Some of the applicatloils of the system are similar to those of Wadler's views [Wad87]. However, our system is considerably more general, and more orthogonal to the underlying language.

[Tiur90] J. Tiuryn. Type Inference Problems – A Survey. *LNCS*, 452:105–120, 1990.

[Tiur92] J. Tiuryn. Subtype Inequalities. In *Proc. Logic in Computer Science 92*, pages 308–315, 1992.

**Abstract:** In this paper we study the complexity of the satisfiability problem for subtype inequalities in simple types. The naive algorithm which solves this problem runs in non-deterministic exponential time for every pre-defined poset of atomic subtypings. In this paper we show that over certain finite posets of atomic subtypings the satisfiability problem for subtype inequalities is PSPACE-hard. On the other hand we prove that if the poset of atomic subtypings is a disjoint union of lattices, then the satisfiability problem for subtype inequalities is solvable in PTIME. This result covers the important special case of the unification problem which can be obtained when the atomic subtype relation is equality (in this case the poset is a union of one-element lattices).

[Trag76] Barry Trager. Algebraic factoring and rational function integration. *Proceedings of SYMSAC'76*, pages 219–226, 1976.

**Abstract:** This paper presents a new, simple, and efficient algorithm for factoring polynomials in several variables over an algebraic number field. The algorithm is then used interatively to construct the splitting field of a polynomial over the integers. Finally the factorization and splitting field algorithms are applied to the problem of determining the transcendental part of the integral of a rational function. In particular, a constructive procedure is given for finding a least degree extension field in which the integral can be expressed.

[Trag84] Barry Trager. *Integration of Algebraic Functions*. PhD thesis, MIT, 1984.

**Abstract:** We show how the "rational" approach for integrating algebraic functions can be extended to handle elementary functions. The resulting algorithm is a practical decision procedure for determining whether a given elementary function has an elementary antiderivative, and for computing it if it exists.

**Link:** http://www.dm.unipi.it/pages/gianni/public_html/Alg-Comp/thesis.pdf

[Turn85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. *Lecture Notes in Computer Science*, 201:1–16, 1985.

**Link:** http://miranda.org.uk/nancy.html

[Turn86] D. A. Turner. An Overview of Miranda. *SIGPLAN Notices*, 21(12):158–166, 1986.

**Link:** http://miranda.org.uk/

[Turn91] Raymond Turner. *Constructive Foundations for Functional Languages*. McGraw-Hill, 1991, 9780077074111.

[Unga91] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. *Lisp and Symbolic Computation*, 4(3), 1991.

**Abstract:** SELF is an object-oriented language for exploratory programming based on a small number of simple and concrete ideas:

prototypes, slots, and behavior. Prototypes combine inheritance and instantiation to provide a framework that is simpler and more flexible than most object-oriented languages. Slots unite variables and procedures into a single construct. This permits the inheritance hierarchy to take over the function of lexical scoping in conventional languages. Finally, because SELF does not distinguish state from behavior, it narrows the gaps between ordinary objects, procedures, and closures. SELF s simplicity and expressiveness offer new insights into object-oriented computation.

[Vars81] Gamkrelidze Varshamov. Method of construction of primitive polynomials over finite fields. *Soobsheh. Akad. Nauk Gruzin.*, 99:61–64, 1981.

[Volp91] Dennis M. Volpano and Geoffrey S. On the Complexity of ML Typability with Overloading. Technical Report TR91-1210, Cornell University, 1991.

**Abstract:** We examine the complexity of type checking in an ML-style type system that permits functions to be overloaded with different types. In particular, we consider the extension of the ML Type system proposed by Wadler and Blott in the appendix of [WB89], with global overloading only, that is, where the only overloading is that which exists in an initial type assumption set; no local overloading via over and inst expressions is allowed. It is shown that under a correct notion of well-typed terms, the problem of determining whether a term is well typed with respect to an assumption set in this system is undecidable. We then investigate limiting recursion in assumption sets, the source of the undecidability. Barring mutual recursion is considered, but this proves too weak, for the problem remains undecidable. Then we consider a limited form of recursion called parametric recursion. We show that although the problem becomes decidable under parametric recursion, it appears harder than conventional ML typability, which is complete for DEXPTIME [Mai90].

[Wadl88] Philip Wadler and Stephen Blott. How to Make Ad-hoc Polymorphism Less Ad hoc. In *Proc 16th ACM SIGPLAN-SIGACT Symp. on Princ. of Prog. Lang*, pages 60–76, 1988, 0-89791-294-2.

**Abstract:** This paper presents *type classes*, a new approach to *ad-hoc* polymorphism. Type classes permit overloading of arithmetic operators such as multiplication, and generalise the "eqtype variables" of Standard ML Type classes extend the Hindley/Milner polymorphic type system, and provide a new approach to issues that arise in object-oriented programming, bounded type quantification, and abstract data types. This paper provides an informal introduction to type classes, and defines them formally by means of type inference rules

**Link:** http://202.3.77.10/users/karkare/courses/2010/cs653/Papers/ad-hoc-polymorphism.p

[Wald92] Uwe Waldmann. Semantics of Order-sorted Specifications. *Theoretical Computer Science*, 94(1-2):1–35, 1992.

**Abstract:** Order-sorted specifications (i.e. many-sorted specifications with subsort relations) have been proved to be a useful tool for the description of partially defined functions and error handling in

abstract data types. Several definitions for order-sorted algebras have been proposed. In some papers an operator symbol, which may be multiply declared, is interpreted by a family of functions (overloaded algebras). In other papers it is always interpreted by a single function (nonoverloaded algebras). On the one hand, we try to demonstrate the differences between these two approaches with respect to equality, rewriting and completion; on the other hand, we prove that in fact both theories can be studied in parallel provided that certain notions are suitably defined. The overloaded approach differs from the many-sorted and the nonoverloaded one in that the overloaded term algebra is not necessarily initial. We give a decidable sufficient criterion for the initiality of the term algebra, which is less restrictive than GJM-regularity as proposed by Goguen, Jouannaud and Meseguer. Sort-decreasingness is an important property of rewrite systems since it ensures that confluence and Church-Rosser property are equivalent, that the overloaded and nonoverloaded rewrite relations agree, and that variable overlaps do not yield critical pairs. We prove that it is decidable whether or not a rewrite rule is sort-decreasing, even if the signature is not regular. Finally, we demonstrate that every overloaded completion procedure may also be used in the nonoverloaded world, but not conversely, and that specifications exist that can only be completed using the nonoverloaded semantics.

[Wand87] Mitchell Wand. Complete Type Inference for Simple Objects. In *Symp. on Logic in Computer Science*, pages 22–25, 1987.

**Abstract:** The problem of strong typing is considered for a model of object-oriented programming systems. These systems permit values which are records of other values, and in which fields inside these records are retrieved by name. A type system is proposed that permits classification of these kinds of values and programs by the type of their result, as is usual in strongly-typed programming languages. The type system has two important properties: it admits multiple inheritance, and it has a syntactically complete type inference system.

[Wand88] Mitchell Wand. Corrigendum: Complete Type Inference for Simple Objects. In *Symp. on Logic in Computer Science*, pages 5–8, 1988.

**Abstract:** An error has been pointed out in the author's paper (see Proc. 2nd IEEE Symp. on Logic in Computer Science, p 37-44 (1987)). It appears that there are programs without principal type schemes in the system in that paper.

[Wand89] Mitchell Wand. Type Inference for Record Concatenation and Multiple Inheritance. In *Logic in Computer Science*, 1989, 0-8186-1954-6.

**Abstract:** The author shows that the type inference problem for a lambda calculus with records, including a record concatenation operator, is decidable. He shows that this calculus does not have principal types but does have finite complete sets of type, that is, for any term M in the calculus, there exists an effectively generable finite set of type schemes such that every typing for M is an instance of one of the schemes in the set. The author shows how a simple model of object-

oriented programming, including hidden instance variables and multiple inheritance, may be coded in this calculus. The author concludes that type inference is decidable for object-oriented programs, even with multiple inheritance and classes as first-class values.

[Wand91] Mitchell Wand. Type Inference for Record Concatenation and Multiple Inheritance. *Information and Computation*, 93:1–15, 1991.

**Abstract:** We show that the type inference problem for a lambda calculus with records, including a record concatenation operator, is decidable. We show that this calculus does not have principal types, but does have finite complete sets of types: that is, for any term M in the calculus, there exists an effectively generable finite set of type schemes such that every typing for M is an instance of one of the schemes in the set. We show how a simple model of object-oriented programming, including hidden instance variables and multiple inheritance, may be coded in this calculus. We conclude that type inference is decidable for object-oriented programs, even with multiple inheritance and classes as first-class values.

[Wass89] A. Wassermann. Konstruktion von Normalbasen. *Bayreuther Math. Schriften*, 31:1–9, 1989.

[Watt94a] S.M. Watt, P.A. Broadbery, S.S. Dooley, P. Iglio, J.M. Steinbach, S.C. Morrison, and R.S. Sutor. AXIOM Library Compiler Users Guide, 1994.

[Webe05] Andreas Weber. A Type-Coercion Problem in Computer Algebra. In *Artificial Intelligence and Symbolic Mathematical Computing*, Lecture Notes in Computer Science 737, pages 188–194. Springer, 2005.

**Abstract:** An important feature of modern computer algebra systems is the support of a rich type system with the possibility of type inference. Basic features of such a system are polymorphism and coercion between types. Recently the use of order-sorted rewrite systems was proposed as a general framework. We will give a quite simple example of a family of types arising in computer algebra whose coercion relations cannot be captured by a finite set of first-order rewrite rules.

[Webe92b] Andreas Weber. Structuring the Type System of a Computer Algebra System. Technical report, Wilhelm-Schickard-Institut fur Informatik, 1992.

**Abstract:** Most existing computer algebra systems are pure symbol manipulating systems without language support for the occuring types. This is mainly due to the fact taht the occurring types are much more complicated than in traditional programming languages. In the last decade the study of type systems has become an active area of research. We will give a proposal for a type system showing that several problems for a type system of a symbolic computation system can be solved by using results of this research. We will also provide a variety of examples which will show some of the problems that remain and that will require further research.

**Link:** http://cg.cs.uni-bonn.de/personal-pages/weber/publications/pdf/WeberA/Weber92a.p

[Webe93b] Andreas Weber. *Type Systems for Computer Algebra.* PhD thesis, University of Tubingen, 1993.

**Abstract:** We study type systems for computer algebra systems, which frequently correspond to the "pragmatically developed" typing constructs used in AXIOM. A central concept is that of type classes which correspond to AXIOM categories. We will show that types can be syntactically described as terms of a regular order-sorted signature if no type parameters are allowed. Using results obtained for the functional programming language Haskell we will show that the problem of type inference is decidable. This result still holds if higher-order functions are present and parametric polymorphism is used. These additional typing constructs are useful for further extensions of existing computer algebra systems: These typing concepts can be used to implement category theoretic constructs and there are many well known constructive interactions between category theory and algebra. On the one hand we will show that there are well known techniques to specify many important type classes algebraically, and we will also show that a formal and algorithmically Feasible treatment of the interactions of algebraically specified data types and type classes is possible. On the other hand we will prove that there are quite elementary examples arising in computer algebra which need very "strong" formalisms to be specified and are thus hard to handle algorithmically. We will show that it is necessary to distinguish between types and elements as parameters of parameterized type classes. The type inference problem for the former remains decidable whereas for the latter it becomes undecidable. We will also show that such a distinction can be made quite naturally. Type classes are second-order types. Although we will show that there are constructions used in mathematics which imply that type classes have to become first-order types in order to model the examples naturally, we will also argue that this does not seem to be the case in areas currently accessible for an algebra system. We will only sketch some systems that have been developed during the last years in which the concept of type classes as first-order types can be expressed. For some of these systems the type inference problem was proven to be undecidable. Another fundamental concept for a type system of a computer algebra system at least for the purpose of a user interface are coercions. We will show that there are cases which can be modeled by coercions but not by an "inheritance mechanism", i. e. the concept of coercions is not only orthogonal to the one of type classes but also to more general formalisms as are used in object-oriented languages. We will define certain classes of coercions and impose conditions on important classes of coercions which will imply that the meaning of an expression is independent of the particular coercions that are used in order to type it. We shall also impose some conditions on the interaction between polymorphic operations defined in type classes and coercions that will yield a unique meaning of an expression independent of the type which is assigned to it if coercions are present there will very frequently be several possibilities to assign types to expressions. Often it is not only possible to coerce one type into another but it will be the case that two types are actually isomorphic . We will show that isomorphic types have properties that

cannot be deduced from the properties of coercions and will shortly discuss other possibilities to model type isomorphisms. There are natural examples of type isomorphisms occurring in the area of computer algebra that have a "problematic" behavior. So we will prove for a certain example that the type isomorphisms cannot be captured by a finite set of coercions by proving that the naturally associated equational theory is not finitely axiomatizable. Up to now few results are known that would give a clear dividing line between classes of coercions which have a decidable type inference problem and classes for which type inference becomes undecidable. We will give a type inference algorithm for some important classes of coercions. Other typing constructs which are again quite orthogonal to the previous ones are those of *partial functions* and of types *depending on elements* . We will link the treatment of *partial functions* in AXIOM to the one used in order-sorted algebras and will show some problems which arise if a seemingly more expressive solution were used. There are important cases in which *types depending on elements* arise naturally. We will show that not only type inference but even type checking is undecidable for relevant cases occurring in computer algebra.

[Webe95] Andreas Weber. On coherence in computer algebra. *J. Symb. Comput.*, 19(1-3):25–38, 1995.

**Abstract:** Modern computer algebra systems (e.g. AXIOM) support a rich type system including parameterized data types and the possibility of implicit coercions between types. In such a type system it will be frequently the case that there are different ways of building coercions between types. An important requirement is that all coercions between two types coincide, a property which is called *coherence*. We will prove a coherence theorem for a formal type system having several possibilities of coercions covering many important examples. Moreover, we will give some informal reasoning why the formally defined restrictions can be satisfied by an actual system.

**Link:** http://cg.cs.uni-bonn.de/personal-pages/weber/publications/pdf/WeberA/Weber94e.p

[Weil71] André Weil. Courbes algébriques et variétés Abeliennes, 1971.

[Weis88] V. Weispfenning. The complexity of linear problems in fields. *J. of Symbolic Computation*, 5(1-2):3–27, 1988.

**Abstract:** We consider linear problems in fields, ordered fields, discretely valued fields (with finite residue field or residue field of characteristic zero) and fields with finitely many independent orderings and discrete valuations. Most of the fields considered will be of characteristic zero. Formally, linear statements about these structures (with parameters) are given by formulas of the respective first-order language, in which all bound variables occur only linearly. We study symbolic algorithms (*linear elimination procedures*) that reduce linear formulas to linear formulas of a very simple form, i.e. quantifier-free linear formulas, and algorithms (*linear decision procedures*) that decide whether a given linear sentence holds in all structures of the given class. For all classes of fields considered, we find linear elimina-

tion procedures that run in double exponential space and time. As a consequence, we can show that for fields (with one or several discrete valuations), linear statements can be transferred from characteristic zero to prime characteristic $p$, provided $p$ is double exponential in the length of the statement. (For similar bounds in the non-linear case, see Brown, 1978.) We find corresponding linear decision procedures in the Berman complexity classes $\bigcup_{c\in N} STA(*, 2^{cn}, dn)$ for $d = 1, 2$. In particular, all these procedures run in exponential space. The technique employed is quantifier elimination via Skolem terms based on Ferrante and Rackoff (1975). Using ideas of Fischer and Rabin (1974), Berman (1977), Frer (1982), we establish lower bounds for these problems showing that our upper bounds are essentially tight. For linear formulas with a bounded number of quantifiers all our algorithms run in polynomial time. For linear formulas of bounded quantifier alternation most of the algorithms run in time $2^{O(n^k)}$ for fixed $k$.

**Link:** http://www.sciencedirect.com.proxy.library.cmu.edu/science/article/pii/S07477171

[Weis92] V. Weispfenning. Comprehensive Groebner bases. *J. Symbolic Computation*, 14(1):1–29, 1992.

**Abstract:** Let $K$ be an integral domain and let $S$ be the polynomial ring $K[U_1, \ldots, U_m; X_1, \ldots, X_n]$. For any finite $F \subseteq S$, we construct a comprehensive Groebner basis of the ideal $Id(F)$, a finite ideal basis of $Id(F)$ that is a Groebner basis of $Id(F)$ in $K'[X_1, \ldots, X_n]$ for every specialization of the parameters $U_1, \ldots, U_m$ in an arbitrary field $K^1$. We show that this construction can be performed with the same worst case degree bounds in the main variable $X_i$, as for ordinary Groebner bases; moreover, examples computed in an ALDES/SAC-2 implementation show that the construction is of practical value. Comprehensive Groebner bases admit numerous applications to parametric problems in algebraic geometry; in particular, they yield a fast elimination of quantifier-blocks in algebraically closed fields

[Weis94] V. Weispfenning. Quantifier elimination for real algebra the cubic case. In *Proc ISSAC'94*, pages 258–263. ACM, 1994, 0-89791-638-7.

**Abstract:** We present a special purpose quantifier elimination method that eliminates a quantifier $\exists x$ in formulas $\exists x(\rho)$ where $\rho$ is a boolean combination of polynomial inequalities of degree $\leq 3$ with respect to $x$. The method extends the virtual substitition of parameterized test points developed in [Weispfenning 1, Loos and Weispf.] for the linear case and in [Weispfenning 2] for the quadratic case. It has similar upper complexity bounds and offers similar advantages (relatively large preprocessing part, explicit parametric solutions). Small examples suggest that the method will be of practical significance.

[Weis97] V. Weispfenning. Quantifier elimination for real algebra - the quadratic case and beyond. *Applicable Algebra in Engineering, Communication and Computing*, 8(2):85–101, 1997.

**Abstract:** We present a new, "elementary" quantifier elimination method for various special cases of the general quantifier elimination problem for the first-order theory of real numbers. These include

the elimination of one existential quantifier $\exists x$ in front of quantifier-free formulas restricted by a non-trivial quadratic equation in $x$ (the case considered also in [7]), and more generally in front of arbitrary quantifier-free formulas involving only polynomials that are quadratic in $x$. The method generalizes the linear quantifier elimination method by virtual substitution of test terms in [9]. It yields a quantifier elimination method for an arbitrary number of quantifiers in certain formulas involving only linear and quadratic occurrences of the quantified variables. Moreover, for existential formulas $\phi$ of this kind it yields sample answers to the query represented by $\phi$. The method is implemented in REDUCE as part of the REDLOG package (see [4,5]). Experiments show that the method is applicable to a range of benchmark examples, where it runs in most cases significantly faster than the QEPCAD package of Collins and Hong. An extension of the method to higher degree polynomials using Thoms lemma is sketched.

**Link:** https://link-springer-com.proxy.library.cmu.edu/article/10.1007%2Fs002000050055

[Weis98] V. Weispfenning. A New Approach to Quantifier Elimination for Real Algebra. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Springer, 1998, 3-211-82794-3.

**Abstract:** Quantifier elimination for the elementary formal theory of real numbers is a facinating area of research at the intersection of various field of mathematics and computer science, such as mathematical logic, commutative algebra and algebraic geometry, computer algebra, computational geometry and complexity theory. Originally the method of quantifier elimination was invented (among others by Th. Skolem) in mathematical logic as a technical tool for solving the decision problem for a formalized mathematical theory. For the elementary formal theory of real numbers (or more accurately of real closed fields) such a quantifier elimination procedure was established in the 1930s by A. Tarski, using an extension of Sturm's theorem of the 1830s for counting the number of real zeros of a univariate polynomial in a given interval. Since then an abundance of new decision and quantifier elimination methods for this theory with variations and optimizations has been published with the aim both of establishing the theoretical complexity of the problem and of finding methods that are of practical importance (see Arnon 1988a and the discussion and references in Renegar 1992a, 1992b, 1992c for a comparison of these methods). For subproblems such as elimination of quantifiers with respect to variables, that are linearly or quadratically restricted, specialized methods have been developed with good success (see Weispfenning 1988, Loos and Weispfenning 1993; Hong 1992d; Weispfenning 1997).

[Whee12] James T. Wheeler. Differential Forms, September 2012.

**Link:** http://www.physics.usu.edu/Wheeler/ClassicalMechanics/CMDifferentialForms.pdf

[Wirs82] Martin Wirsing and Manfred Broy. An Analysis of Semantic Models for Algebraic Specifications. In *Theoretical Foundations of Programming Methodology*, pages 351–413. Springer, 1982, 978-94-009-7893-5.

**Abstract:** Data structures, algorithms and programming languages can be described in a uniform implementation-independent way by axiomatic abstract data types i.e. by algebraic specifications defining abstractly the properties of objects and functions. Different semantic models such as initial and terminal algebras have been proposed in order to specify the meaning of such specifications -often involving a considerable amount of category theory. A more concrete semantics encompassing these different approaches is presented: Abstract data types are specified in hierarchies, employing "primitive" types on which other types are based. The semantics is defined to be the class of all partial heterogeneous algebras satisfying the axioms and respecting the hierarchy. The interpretation of a specification as its initial or terminal algebra is just a constraint on the underlying data. These constraints can be modified according to the specification goals. E.g. the data can be specified using total functions; for algorithms partial functions with syntactically checkable domains seem appropriate whereas for programming languages the general notion of partiality is needed, Model-theoretic and deduction-oriented conditions are developed which ensure properties leading to criteria for the soundness and complexity of specifications. These conditions are generalized to parameterized types, i.e. type procedures mapping types into types. Syntax and different semantics of parameter are defined and discussed. Criteria for proper parameterized specifications are developed. It is shown that the properties of proper specifications viz. of snowballing and impeccable types are preserved under application of parameterized types finally guaranteeing that the composition of proper small specifications always leads to a proper large specification.

[Wirs91] Martin Wirsing. Algebraic Specification. In *Handbook of Theoretical Computer Science (Vol B)*, chapter 13, pages 675–788. MIT Press, 1991, 0-444-88074-7.

[Wirt83] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1983, 978-3-642-96878-5.

[Wirt88] N. Wirth. Type Extensions. *TOPLAS*, 10(2):203–214, 1988.

**Abstract:** Software systems represent a hierarchy of modules. Client modules contain sets of procedures that extend the capabilities of imported modules. This concept of extension is here applied to data types. Extended types are related to their ancestor in terms of a hierarchy. Variables of an extended type are compatible with variables of the ancestor type. This scheme is expressed by three language constructs only: the declaration of extended record types, the type test, and the type guard. The facility of extended types, which closely resembles the class concept, is defined in rigorous and concise terms, and an efficient implementation is presented.

[Wolf91] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, 1991, 978-0201515022.

[Yun76] D.Y.Y Yun. On square-free decomposition algorithms. In *Proceedings of SYMSAC'76*, pages 26–35, 1976.

[Zari75] Oscar Zariski and Pierre Samuel. *Commutative Algebra*. Graduate Texts in Mathematics. Springer-Verlag, 1975, 978-0387900896.

[Zipp93] Richard Zippel. The Weyl Computer Algebra Substrate. In *Design and Implementation of Symbolic Computation Systems '93*, DISCO 93, page 303=318, 1993.

**Abstract:** Weyl is a new type of computer algebra substrate that extends an existing, object oriented programming language with symbolic computing mechanisms. Rather than layering a new language on top of an existing one, Weyl behaves like a powerful subroutine library, but takes heavy advantage of the ability to overload primitive arithmetic operations in the base language. In addition to the usual objects manipulated in computer algebra systems (polynomial, rational functions, matrices, etc.), domains (e.g., Z, Q[x, y, z]) are also first class objects in Weyl.

[Zuck19] Philip Zucker. Grobner Bases and Optics, 2019.

**Link:** http://www.philipzucker.com/grobner-bases-and-optics

# Index