Axiom / FriCAS

Christoph Koutschan

Research Institute for Symbolic Computation
Johannes Kepler Universität Linz, Austria

Computer Algebra Systems
15.11.2010

Master's Thesis: The ISAC project

- initiative at Graz University of Technology
  - Institute for Software Technology
  - Institute for Information Systems and Computer Media
- experimental software assembling open source components with as little glue code as possible
- feasibility study for a novel kind of transparent single-stepping software for applied mathematics
- experimenting with concepts and technologies from
  - computer mathematics (theorem proving, symbolic computation, model based reasoning, etc.)
  - e-learning (knowledge space theory, usability engineering, computer-supported collaboration, etc.)
- The development employs academic expertise from several disciplines. The challenge for research is interdisciplinary cooperation.

Rewriting, a basic CAS technique

This technique is used in simplification, equation solving, and many other CAS functions, and it is intuitively comprehensible. This would make rewriting useful for educational systems—if one copes with the problem, that even elementary simplifications involve hundreds of rewrites.

As an example see:
http://www.ist.tugraz.at/projects/isac/www/content/publications.html#DA-M02-main

“Reverse rewriting” for comprehensible justification

Many CAS functions can *not* be done by rewriting, for instance cancelling multivariate polynomials, factoring or integration. However, respective inverse problems can be done by rewriting and produce human readable derivations.

As an example see:
http://www.ist.tugraz.at/projects/isac/www/content/
publications.html#GGTs-von-Polynomen

Equation solving made transparent

Re-engineering equation solvers in “transparent single-stepping systems” leads to types of equations, arranged in a tree. ISAC’s tree of equations are to be compared with what is produced by tracing facilities of Mathematica and/or Maple. How could ISAC’s equation solver be extended?

See also:
http://www.ist.tugraz.at/projects/isac/www/content/publications.html#da-mlang

ISAC, a transparent single-stepping system

What distinguishes ISAC from a CAS? What are the advantages of a system based on a computer theorem prover (CTP)? What novel kinds of services can such a system provide for education?

See also:
https://lsiit-cnrs.unistra.fr/DG-Proofs-Construction/index.php/ISAC_system
http://www.ist.tugraz.at/projects/isac

CAS functionality adopted by CTP

There are good reasons for warning “never trust a CAS”. Computer theorem provers (CTP), however, allow users to trust the correctness of their results. Now, since more and more CAS functionality is taken over by CTP — how can such trust be ensured?

See also:
http://www.score.cs.tsukuba.ac.jp/˜kaliszyk/docs/ck_thesis_webdoc.pdf

Axiom

- "The Scientific Computation System"
- general-purpose computer algebra system
- freely available from http://www.axiom-developer.org
- Axiom is a literate program

History

- “Scratchpad”, developed since 1971 at IBM
- key developers: Richard Dimick Jenks, Barry Trager, Stephen Watt, James Davenport, Robert Sutor, Scott Morrison
- sold to the NAG (Numerical Algorithms Group) in the 1990s and renamed to “Axiom”
- 2001: commercial activity was abandoned and Axiom was distributed as open source
- 2007: split into two branches: OpenAxiom and FriCAS

Literate Programming

- programming paradigm introduced by Donald Knuth
- program and documentation are woven together
- like an essay in natural language
- contains macros and code snippets

Design

- strongly typed
- interpreter language: SPAD
- extension languages: A# and Aldor
- every object has a type
- types are in a mathematically correct type hierarchy

Types

- types in Axiom are dynamic objects, i.e., they are created at runtime
- algebraic types: polynomials, matrices, power series, etc.
- types as data structures: lists, dictionaries, input files, etc.
- types combine in any meaningful way, e.g., polynomials of matrices, etc.

Categories

- define algebraic properties
- ensure mathematical correctness, e.g., don't allow to build matrices of input files
- through categories, programs can discover certain properties, e.g.,
  - polynomials of continued fractions are commutative
  - polynomials of matrices are not commutative
- categories allow to define algorithms in their most general setting, e.g., Euclid's gcd algorithm can be implemented such that it works over any Euclidean domain

Starting Axiom (FriCAS)

```
ckoutsch@gonzales:~$ axiom
Checking for foreign routines
AXIOM="/zvol/fricas/x86_64/lib/fricas/target/x86_64-unknown
spad-lib="/zvol/fricas/x86_64/lib/fricas/target/x86_64-unkn
foreign routines found
openServer result 0
        FriCAS (AXIOM fork) Computer Algebra System
                   Version: FriCAS 1.0.9
        Timestamp: Monday June 7, 2010 at 12:22:18
-------------------------------------------------------
  Issue )copyright to view copyright notices.
  Issue )summary for a summary of useful system commands.
  Issue )quit to leave FriCAS and return to shell.
-------------------------------------------------------


(1) ->
```

Numbers (1)

```
(1) -> 8

   (1)  8
                    Type: PositiveInteger
(2) -> -8

   (2)  - 8
                    Type: Integer
(3) -> 2/3

        2
   (3)  -
        3           Type: Fraction(Integer)

(4) -> 2/3 :: Float
   (4)  0.6666666666 6666666667
                    Type: Float
```

Numbers (2)

```
(5) -> factor(5040)

         4 2
   (5)  2 3 5 7
                             Type: Factored(Integer)
(6) -> % * 121

         4 2       2
   (6)  2 3 5 7 11
                             Type: Factored(Integer)
(7) -> roman(1998)

   (7)  MCMXCVIII
                             Type: RomanNumeral
(8) -> % + 2
   (8)  MM
                             Type: RomanNumeral
```

More complicated numbers

```
(9) -> decimal(1/352)

                --
   (9)  0.0028409
                         Type: DecimalExpansion
(10) -> (2 + sqrt 63)^(1/3)
          +----------+
         3|  +-+
   (10)  \|3\|7  + 2
                         Type: AlgebraicNumber
(11) -> factor(89 - 23*%i)
                              2         2
   (11)  - (1 + %i)(2 + %i) (3 + 2%i)
                         Type: Factored(Complex(Integer))

(12) -> a : PrimeField(7) := 13
   (12)  6
                         Type: PrimeField(7)
```

Polynomials (1)

```
(3) -> x^6
         6
   (3)  x                   Type: Polynomial(Integer)
(4) -> x^(-6)
         1
   (4)  --
         6
        x                   Type: Fraction(Polynomial(Integer))
(5) -> x/6
        1
   (5)  - x
        6                   Type: Polynomial(Fraction(Integer))

(6) -> % :: Fraction Polynomial Integer
        x
   (6)  -
        6                   Type: Fraction(Polynomial(Integer))
```

Polynomials (2)

```
(7) -> (x+y)^3

          3        2      2      3
   (7)   y  + 3x y  + 3x y + x
                    Type: Polynomial(Integer)
(8) -> (x+y)^3 / (x+y)

          2           2
   (8)   y  + 2x y + x
                    Type: Fraction(Polynomial(Integer))
(9) -> % :: Polynomial(Integer)

          2           2
   (9)   y  + 2x y + x
                    Type: Polynomial(Integer)
```

Coercion

- an automatic transformation of an object of one type to an object of a similar target type
- can be done explicitly by the infix operation
  `object ::  type`
- is done automatically by the interpreter when a type mismatch occurs

Simple Assignments

```
(1) -> x := 4
   (1)  4
                             Type: PositiveInteger
(2) -> x := x + z
   (2)  z + 4
                             Type: Polynomial(Integer)
(3) -> x : Integer
   You cannot declare x to be of type Integer because
      either the declared type of x or the type of the
      value of x is different from Integer .

(3) -> y : Integer
                             Type: Void
(4) -> y := 5/6
   Cannot convert right-hand side of assignment to an
      object of the type Integer of the left-hand side.
```

The Fibonacci example

```
(10) -> fib(0) == 0
                                 Type: Void
(11) -> fib(1) == 1
                                 Type: Void
(12) -> fib(n) == fib(n-1) + fib(n-2)
                                 Type: Void
(13) -> fib(90)
   Compiling function fib with type
        Integer -> NonNegativeInteger
   Compiling function fib as a recurrence relation.

   (13)  2880067194370816120
                                 Type: PositiveInteger
```

Data structures: Stream

```
(7) -> s := [fib(i) for i in 0..]
   Compiling function fib with type
        Integer -> NonNegativeInteger
   Compiling function fib as a recurrence relation.

   (7)  [0,1,1,2,3,5,8,13,21,34,...]
                            Type: Stream(NonNegativeInteger)
(8) -> s.1

   (8)  0
                            Type: NonNegativeInteger
(9) -> s.100

   (9)  218922995834555169026
                            Type: PositiveInteger
```

Data structures: Series

```
24) -> s := series(1/(1-x)^3, x=0)

   (24)
                     2        3        4        5        6        7
      1 + 3x + 6x  + 10x  + 15x  + 21x  + 28x  + 36x

                 8        9        10        11
        + 45x  + 55x  + 66x    + O(x   )

     Type: UnivariatePuiseuxSeries(Expression(Integer),x,0)

(25) -> coefficient(s, 200)

   (25)  20301
                                   Type: Expression(Integer)
```

Data structures: Series

```
(26) -> exp(s)
   (26)
                        21%e   2    65%e   3    747%e   4    10161%e   5
     %e + 3%e x + ---- x   + ---- x   + ----- x   + ------- x
                          2          2          8            40
   +
     931467%e   7    3634053%e   8    18497567%e   9    100196630
     -------- x   + --------- x   + ---------- x   + ---------
        560              896              1920               44800
   Type: UnivariatePuiseuxSeries(Expression(Integer),x,0)

28) -> D(s)
   (28)
                        2       3         4         5         6         7
     3 + 12x + 30x  + 60x  + 105x  + 168x  + 252x  + 360x
   +
          10        11
```

Polymorphic Algorithms

```
(1) -> sys := [3*x^3+y+1=0, y^2=4]
                  3          2
   (1)  [y + 3x  + 1= 0,y  = 4]
   Type: List(Equation(Polynomial(Integer)))

(2) -> solve(sys, 10^(-20))

                         2046441276570763724449
   (2)  [[y= - 2,x= ---------------------],[y= 2,x= - 1]]
                         2951479051793528258 56
   Type: List(List(Equation(Polynomial(Fraction(Integer)))

(3) -> solve(sys, 1.e-20)
   (3)  [[y= - 2.0,x= 0.6933612743 5063470484],
         [y= 2.0,x= - 1.0]]
   Type: List(List(Equation(Polynomial(Float))))
```

Polymorphic Algorithms

```
(4) -> radicalSolve(sys)
   (4)                               +---+                   +---+
                                  - \|- 3  + 1              \|- 3
   [[y= 2,x= - 1], [y= 2,x= ------------], [y= 2,x= -------
                                      2                       2
                                       +---+ +-+
                 1                    \|- 1 \|3  - 1
    [y= - 2,x= ----], [y= - 2,x= --------------], [y= - 2,x
               3+-+                          3+-+
               \|3                          2\|3
    Type: List(List(Equation(Expression(Integer))))

(5) -> solve(sys)
                                   2                       3
   (5)  [[y= 2,x= - 1],[y= 2,x  - x + 1= 0],[y= - 2,3x  - 
    Type: List(List(Equation(Fraction(Polynomial(Integer))
```

Design principles (1)

Types are defined by abstract datatype programs.

- basic types are called *domains of computation* ("domains")
- definition of a domain:
  `Name(...):  Exports == Implementation`
- the capitalized `Name` refers to the class of its members, e.g., `Integer` denotes the "class of integers"
- parameters (in parentheses) are usually domains again, e.g., `Matrix(Integer)` denotes "matrices over the integers" or `List(Matrix(Polynomial(Integer)))`
- `Exports` specifies operations for creating and manipulating objects of the domain
- `Integer` exports 0, 1, +, −, and ∗

Design principles (2)

The type of basic objects is a domain or subdomain.

- every Axiom object belongs to a unique domain
- the domain of an object is also called its type
- subdomain: a domain with a "membership predicate", e.g., `PositiveInteger` is a subdomain of `Integer` with the predicate "> 0"
- subdomains are defined by abstract datatype programs, similar to domains
  - the `Exports` part must be a subset of the exports of the domain
  - the `Implementation` part optionally gives special definitions

Design principles (3)

Domains have types called categories.

- the type of a domain or subdomain is called a *category*
- Note: category here doesn't coincide with the algebraic concept of a category
- categories are described by programs of the form `Name(...):  Category == Exports`
- `Name` designates a class of domains, e.g., the category `Ring` designates the class of all rings.
- the type of every category is the distinguished symbol `Category`
- the `Exports` part defines a set of operations, e.g., the category `Ring` exports `0`, `1`, `+`, `-`, and `*`
- categories serve to ensure type-correctness, e.g., the definition `Matrix(R: Ring)` allows matrices of polynomials, but not matrices of lists.

Domains and Categories (1)

```
(1) -> Integer

   (1)  Integer
                                    Type: Type
(2) -> Ring

   (2)  Ring
                                    Type: Category
(3) -> Integer has Ring

   (3)  true
                                    Type: Boolean

(4) -> Fraction(Integer) has Field

   (4)  true
                                    Type: Boolean
```

Domains and Categories (2)

```
(5) -> Polynomial(Matrix(Integer))

   Polynomial(Matrix(Integer)) is not a valid type.

(5) -> Matrix(Integer) has Ring

   (5)  false
                                    Type: Boolean

(6) -> SquareMatrix(3,Integer) has Ring

   (6)  true
                                    Type: Boolean
```

Definition of Categories

The definition of a category has the syntax:
CategoryForm `:` `Category ==` *Extensions* `[ with` *Exports* `]`

Example:

```
SetCategory(): Category ==
  Join(Type, CoercibleTo OutputForm) with
    "=" : ($,$) -> Boolean
```

- `Type` is the category of all domains
- all exports are constants or operations with type “mapping”
- `$` denotes “this domain”
- `SetCategory` is now “the category of all domains that are sets”.

Extension of Categories

Categories form a hierarchy by extension.

Example:

```
SemiGroup(): Category == SetCategory with
  "*": ($,$) -> $
  "**": ($,PositiveInteger) -> $
```

- each category (except `Type`) has one or more parents
- the parents are mentioned before the `with` part
- any domain can be used in a signature
- default definitions (e.g., for `**`) can be given

Design principles (4)

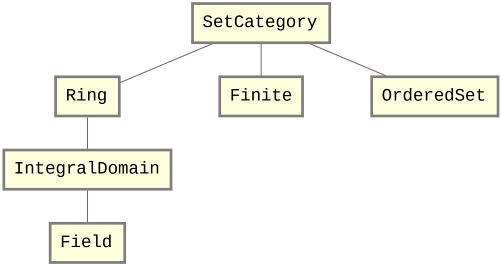
Operations can refer to abstract types.

- all operations have prescribed source and target types
- symbolic domains:
  ```
  R: Ring
  power:  (R, NonNegativeInteger):  R -> R
  power(x,n) == x**n
  ```
- using symbolic domains, algorithms can be programmed in their most general setting

Design principles (5)

Categories form hierarchies.

- categories form directed, acyclic graphs
- Example:



Design principles (6)

Domains belong to categories by assertion.

- each domain must assert which categories it belongs to
- for example, the `Exports` part in `Integer` reads
  `Join(OrderedSet, IntegralDomain, ...)  with ...`
- `Integer` inherits `0`, `1`, `+`, `-`, and `*` from `Ring`
- assertions can be conditional, e.g.
  `Complex(R)` defines its exports by:
  `Ring with ...  if R has Field then Field ...`
- thus `Complex(Float)` is a field, but `Complex(Integer)` is not.

Design principles (7)

Packages are clusters of polymorphic operations.

- a package is a special kind of domain
- for example, define some algorithms for solving equations of polynomials over an arbitrary field `F`:
  ```
  MySolve(F: Field):  Exports == Implementation
  ```
- the `Exports` part specifies the solve operations that the programmer wants to export
- the algorithms then work for any field `F`

Example: Package definition

```
SortPackage(S,A) : Exports == Implementation where
  S: Object
  A: IndexedAggregate(Integer,S)
    with (finiteAggregate; shallowlyMutable)

Exports ==
  bubbleSort!: (A, (S,S)->Boolean) -> A
  insertionSort!: (A, (S,S)->Boolean) -> A
  if S has OrderedSet then
    bubbleSort!: A -> A
    insertionSort!: A -> A

Implementation ==
  bubbleSort!(m,f) ==
    ...
  insertionSort!(m,f) ==
    ...
```

Design principles (8)

The interpreter builds domains dynamically.

- the interpreter reads user input and then builds the needed domains
- Example: create the matrix

$$M = \begin{pmatrix} x^2 + 1 & 0 \\ 0 & x/2 \end{pmatrix}$$

- interpreter builds the domain tower `Matrix`, `Polynomial`, `Fraction`, `Integer`
- operations are called down the tower, and the result is passed back up

Design principles (9)

Axiom code is compiled.

- programs are statically compiled, and then placed into library modules
- static type-checking at compile time

Design principles (10)

Axiom is extensible.

- the entire Axiom library is written in Axiom language
- code-economy
- automatic, guaranteed interaction between old and new code:
  - new algorithm with parameter of type `Field` will work with every field defined in the system (past, present, future)
  - introduce a new field, and it will work as input of any algorithm that requires a field

Influence of Axiom on other CAS

Try `?Domains` in Maple:

. . .

Acknowledgements
Some of the main ideas behind Domains come from the AXIOM system, which was formerly called Scratchpad II. AXIOM also has the notion of parametrized types which it calls domains. The author of Domains would like to acknowledge the use of the primary idea behind AXIOM, namely that of passing as a parameter a collection of functions as a single unit, which the authors of AXIOM have termed a domain.

Literature

Richard D. Jenks, Robert S. Sutor:
AXIOM—The Scientific Computation System