Algorithms and Algorithmic Reasoning in Mathematica

Lectures at the Workshop

"Introduction to Computer Algebra and Applications"

Douala, Cameroon, October 6 - 13, 2017

Bruno Buchberger

Research Institute for Symbolic Computation

Johannes Kepler University, Linz, Austria

Goal of My Lectures

The other lectures: Power of current computer algebra / symbolic computation software systems including the own important contributions of the lecturers.

My lecture:

- **Methodology** of (future) mathematics: automated math verification and invention.
- My current research interest (the “Theorema” project)
- Maybe, impulse for your own research.

(For avoiding frustration: **These lectures are for people who** want to focus on understanding existing algorithmic mathematics and the invention and proof of new algorithmic mathematics. These lectures are not necessary, and maybe not interesting, for people who mainly want to apply existing algorithmic mathematics and math software systems for science, technology, economy ...)

Plan of My Lectures

Develop a **very simple automated reasoner** in the **lectures**.

Tutorial: Play with the reasoner in examples. (Maybe extend the reasoner.)

How Do we “Do” Mathematics? The “Creativity Spiral”

Problem

↓

EXPERIMENTING

↓

Conjecture

↓

PROVING

↓

Theorem

↓

PROGRAMMING

↓

Algorithm

↓

APPLYING

↓

Problem

↓

EXPERIMENTING

↓

Conjecture

↓

PROVING

↓

Theorem

↓

PROGRAMMING

↓

$$\begin{tikzcd}
\text{Algorithm} \arrow[d] \\
\text{APPLYING} \arrow[d] \\
\text{Problem} \arrow[d] \\
\text{EXPERIMENTING} \arrow[d] \\
\text{Conjecture} \arrow[d] \\
\text{PROVING} \arrow[d] \\
\text{Theorem} \arrow[d] \\
\text{PROGRAMMING} \arrow[d] \\
\text{Algorithm} \arrow[d] \\
\text{APPLYING}
\end{tikzcd}$$

......

The “Creativity Spiral”: More Details

Problem (from science, industry, ..., or from “within” mathematics)

↓

EXPERIMENTING (using known algorithms, definitions, theorems, systems...)

↓

Conjecture (as a result of experimenting: new mathematical knowledge in examples,
not yet proved “for all”)

↓

PROVING (one of the two **main mathematical skills**)

↓

Theorem (lemma, proposition, ...; new mathematical knowledge, proved “**for all**”)

↓

PROGRAMMING (the second **main mathematical skill;**
essentially, the correct transition must be proved;
however, in practice, ..., testing?)

↓

Algorithm (method, procedure, program,....)

↓

APPLYING (solve the problem for any instance by applying the algorithm)

↓

Problem

↓

EXPERIMENTING

....

An Example

Problem: Greatest common divisor of two natural numbers.

EXPERIMENTING: 12 and 18 have the same common divisors as 12 and 18-12, ...

Conjecture: for all t, t divides x and y iff t divides x and y - x.

PROVING: exercise.

Theorem (Euclid): for all t, t divides x and y iff t divides x and y - x.

PROGRAMMING: exercise.

Algorithm (Euclid): (for example, in Mathematica: an “algorithmic version” of predicate logic!)

```
Clear[gcd]
gcd[x_, x_] := x
gcd[x_, y_] := gcd[x - y, y] /; x > y
gcd[x_, y_] := gcd[x, y - x] /; y > x
```

APPLYING:

```
gcd[0, 0]
0
```

(A matter of definition!)

```
gcd[15, 15]
15
```

```
gcd[12, 18]
6
```

```
gcd[18, 6]
```

```
6
```

```
gcd[60, 45]
```

```
15
```

```
gcd[45, 60]
```

```
15
```

```
gcd[234 035 435 310, 943 234 035 435] // Timing
```

```
{0.001471, 15}
```

An algorithm for gcd is already built-in in Mathematica

```
GCD[234 035 435 310, 943 234 035 435] // Timing
```

$\{8. \times 10^{-6}, 15\}$

Why is Euclid’s algorithm better than determining the greatest common divisor by factoring?

Why is the built-in algorithm so much faster?

Can Euclid’s algorithm be improved (by going a next round in the “creativity spiral”; see Lehner’s algorithm)?

The Goal of Mathematics on the Object Level and the “Meta” Level

The goal of mathematics: automation **(“algorithmization”)** of solving problems in domains of objects by **thinking** (“proving” and “programming”).

Proving and programming are problems on the **“meta” domain of thoughts (sentences).**

Can we automate **(“algorithmize”) proving and programming (“reasoning”)?**

A Hot Topic

Automation of reasoning is a hot topic today.

Part of "**symbolic computation**" (see my editorial for the Journal of Symbolic Computation, 1985).

Today many catch words are floating around like "artificial intelligence", "machine learning" etc.

There is no upper bound for going higher and higher in the layers of automating mathematical thinking.

In a particular historical situation, the highest layer will always be reserved for human mathematical thinking.

As a consequence, **if you want to stay ahead,** become a master of thinking and a master of automating thinking.

In the Example:

Can we find **algorithms** that,

- given a **conjecture** like the one in the above example, can produce a **proof** (or dis-proof)
- and, given a **problem** like the one in the above example, can produce a correct **algorithm** for the problem?

Algorithmization on the Object Level and the Meta Level in Parallel

This is a recent research theme for which I will give some very simple examples in this lecture.

For this, a language like **Mathematica** (which can be considered as an algorithmic version of “higher-order” predicate logic) is a useful frame.

In my view, **in the future,** the work of mathematicians will be supported by systems in which one can do algorithmic work **on the object level and the meta level in parallel.**

In my **Theorema** Project, I am pursuing this objective since 1995, see for example:

Bruno Buchberger, Tudor Jebelean, Temur Kutsia, Alexander Maletzky, Wolfgang Windsteiger. Theorema 2.0 : Computer - Assisted Natural - Style Mathematics. Journal of Formal Reasoning, 9 (1), pp.149 - 185. 2016. ISSN 1972 - 5787.

In the Theorema system, by similar methods as the ones I will explain in these lectures, for example, the entire theory of Gröbner bases is proved automatically.

Example Theory in the Lecture

“We start from zero” in the literal sense that we take the natural numbers with “zero” and the “successor function” as the only ingredients.

We will show, how we can write a simple automated reasoner for proving simple theorems about simple operations definable over the natural numbers.

Introduction

A Crash Course in Mathematica

The Theory of Natural Numbers

Rough Sketch of an Induction Prover

A More Elaborate Induction Prover

Conclusion

The Main Aspects of Mathematica

Mathematica is a **mathematical software system** designed and built up **by Stephen Wolfram** since 1988.

Mathematica provides a rich **high-level programming language** (the “Wolfram Language”) whose kernel is a version of “higher-order” programming language.

Mathematica also provides a **huge library of algorithms** for all areas of mathematics.

Mathematica is also a **huge knowledge base of facts** from all areas of science and economy. (In an advanced version this is accessible through the **“Wolfram Alpha Machine”.**

Mathematica also provides an **attractive user-interface** that allows to construct nice surface for any package who want to program yourself.

In my lectures at this workshop, **I will focus on the aspect of writing programs in “pattern match” style** both for the **object level and the meta level** of mathematics.

The Documentation Facility of Mathematica

A comprehensive, detailed, and self-explanatory documentation of Mathematica can be accessed via the Menu ‘Help → ‘Wolfram Documentation’.

You then just type a keyword into the find window or click on one of the mathematical subareas displayed and you will obtain entire sequences of examples and tutorials.

For example, if your are interested in machine learning: Search for “machine learning” in the ‘Wolfram Documentation’ and go to the examples in the notebook that pops up:

```
trainingset = {1 → "A", 2 → "A", 3.5 → "B", 4 → "B"};

c = Classify[trainingset]

ClassifierFunction[ Input type: Numerical
                    Classes: A, B ]
```

Use the classifier function to classify a new unlabeled example:

```
c[2.6]

A
```

Obtain classification probabilities for this example:

```
c[2.6, "Probabilities"]

<|A → 0.999618, B → 0.000381686|>
```

.....

Try to understand how to use the Mathematica functions in this section by experimenting with the examples and your own examples.

Calling Algorithms in Mathematica

For basically “all” mathematical problems for which algorithmic solutions are known algorithms are available in Mathematica:

```
Factorial[15]
```

```
1 307 674 368 000
```

```
15 !
```

```
1 307 674 368 000
```

```
Sort[{3, 2, 4, 5, 6, 2}]
```

```
{2, 2, 3, 4, 5, 6}
```

`D[Sin[`a^2`], a]`

`2 a Cos[`a^2`]`

`Integrate[2 a Cos[`a^2`], a]`

`Sin[`a^2`]`

Programming in Mathematica

```
Clear[factorial]
factorial[0] := 1
factorial[n_] := n factorial[n - 1]
```

Tests:

```
Table[factorial[i], {i, 0, 10}]
```

```
{1, 1, 2, 6, 24, 120, 720, 5040, 40 320, 362 880, 3 628 800}
```

```
Clear[factorial]
```

A Representation of the Natural Numbers

We decide to **represent the natural numbers by the Mathematica expressions**

```
0

0⁺
SuperPlus[0]

0⁺⁺
SuperPlus[SuperPlus[0]]

...
```

In “FullForm” these are, actually the expressions

```
0

SuperPlus[0]
SuperPlus[SuperPlus[0]]

...
```

(I do not use the usual notation 0, 1, 2, ... for the natural numbers because they are already built-in in Mathematica and **I want to avoid that anything is already “known” about the natural numbers** except what I will explicitly define.)

A Definition of ‘plus’ on the Natural Numbers

```
Clear[plus]
plus[x_, 0] := x
plus[x_, y_+] := plus[x, y]+
```

(I do not use ‘Plus’ because ‘Plus’ is already built-in in Mathematica. For the same reason, I do not us the usual notation ‘+‘ for ‘plus’.)

Now, we can computer

```
plus[0, 0]
```

```
0
```

```
plus[0++++++, 0++++]
```

$((((((((((0^+)^+)^+)^+)^+)^+)^+)^+)^+)^+$

```
plus[0++++++, 0++++] // FullForm
```

```
SuperPlus[SuperPlus[SuperPlus[
    SuperPlus[SuperPlus[SuperPlus[SuperPlus[SuperPlus[SuperPlus[SuperPlus[0]]]]]]]]]]
```

```
Clear[plus]
```

A Definition of ‘times’ on the Natural Numbers

SEE TUTORIAL

Building up the Theory of Natural Numbers

For building up efficient algorithms for interesting problems that can be expressed in the mathematical structure “natural numbers” (for example the binary or decimal representation of natural numbers and efficient algorithms on these representations) **we need to know many properties of ‘plus’ and ‘times’** and the many other operations (functions and predicates) which we will like to define over the naturals.

Example of a Property of ‘plus’

“**Commutativity**” of ‘plus’:

```
forAll[{x, y}, plus[x, y] ≡ plus[y, x]]
```

(Notation could be different, it only concerns “the surface”. Important: It must be possible to “parse” any expression uniquely in its sub-parts! In Mathematica, the internal form may always be seen by using “FullForm”. We cannot use ‘=’ here for equality because ‘=’ has a specific built-in meaning in Mathematica which would cause the formula to be evaluated in a way we do not want here.)

```
forAll[{x, y}, plus[x, y] ≡ plus[y, x]] // FullForm
forAll[List[x, y], Congruent[plus[x, y], plus[y, x]]]
```

The intended range of the “quantifier” ‘forAll’ is the set consisting of

0

0^{+}

0^{++}

...

and nothing more!

How Can we Prove this Property?

Try it!

We need “induction” over the structure (0, $\square^{+}$). Why? Couldn’t we prove this differently?

“Induction Principle” (Proof Method of Induction)

For any property ‘P’,

in order to prove

```
forAll[{x}, P[x]],
```

it suffices to do the following:

Induction base: Prove

```
P[0].
```

Induction step: Take ‘X’ arbitrary but fix, assume the “induction hypothesis”

```
P[X]
```

and prove

```
P[X⁺].
```

Why does this principle work?

What does “arbitrary but fix” mean?

Let’s Try to Prove Commutativity Now

Since we have two quantified variables, we should look at the formula in the following way:

```
forAll[{x}, forAll[{y}, plus[x, y] ≡ plus[y, x]]]
```

i.e. the formula ‘P’ in the induction would be

```
forAll[{y}, plus[x, y] ≡ plus[y, x]]
```

For the induction basis w.r.t. ‘x’, we hence should prove:

```
forAll[{y}, plus[0, y] ≡ plus[y, 0]]
```

For this, we take y arbitrary but fix, and try to prove

```
plus[0, y] ≡ plus[y, 0].
```

By the definition of ‘plus’ we know

```
plus[y, 0] ≡ y
```

However, from the definition of ‘plus’ we do not yet know that

```
plus[0, y] ≡ y.
```

For this, we need another induction.

Prove “Left Zero”

Thus, before we prove commutativity, let’s prove:

```
forAll[{y}, plus[0, y] ≡ y].
```

Induction base: Prove

```
plus[0, y] ≡ y.
```

This is easy by the first line in the definition of ‘plus’.

Now we take y arbitrary but fix. **Induction hypothesis:**

```
plus[0, y] ≡ y.
```

We have to prove:

$$\text{plus}[0, y^{+}] \equiv y^{+}.$$

In fact, by “simplification” (applying the logic of “equality”) we have:

$$\text{plus}[0, y^{+}] \equiv$$

by the second line in the definition of ‘plus’

$$\text{plus}[0, y]^{+} \equiv$$

by the induction hypothesis

$$y^{+}.$$

The Motivation for Automated Proving

All this is, in principle, **easy but necessary**.

In fact, if we build up a theory like the theory of naturals step by step, by introducing more and more functions and predicates and studying their relationships **we need hundred of such proofs.** This is cumbersome and, of course, also error-prone.

Instead of doing all these proofs individually by “hand” (i.e. by using our brain for each individual proof, let’s better **invest our brain once for writing an algorithm that can do all proofs** in this class by just “pressing a button” (i.e. by giving the properties conjectured - together with the knowledge base of definitions and properties already proved - as an input).

A Rough Sketch of an Induction Prover for Equalities Using Mathematica

First, we represent the definition (“axioms”) from which we start as “rewrite rules”:

```
AxiomsℕPlus = {plus[x_, 0] :→ x, plus[x_, y_⁺] :→ plus[x, y]⁺}
```

```
{plus[x_, 0] :→ x, plus[x_, y_⁺] :→ plus[x, y]⁺}
```

(I will explain the subtle reason for doing this in the lecture.)

The process of simplifying by rewriting, which is the reasoning method to be used for equalities with no induction variables, can then be described in Mathematica like this:

```
Clear[SimplifyByRewriting]
SimplifyByRewriting[expression1_ ≡ expression2_, {equalities___}] :=
 If[
   (expression1 //. {equalities}) === (expression2 //. {equalities}), Proved, Unproved]
```

(I will explain the basic Mathematica functions used in the program in the lecture.)

Test Examples:

```
SimplifyByRewriting[plus[0⁺⁺⁺, 0⁺⁺] ≡ plus[0⁺⁺, 0⁺⁺⁺], AxiomsℕPlus]
```

```
Proved
```

```
SimplifyByRewriting[plus[0⁺⁺⁺, 0⁺⁺] ≡ plus[0⁺⁺⁺⁺, 0⁺⁺⁺], AxiomsℕPlus]
```

```
Unproved
```

Induction for equalities with only one induction variable can then be described in Mathematica like this:

```
Clear[ProveByInduction]

ProveByInduction[forAll[{n1_}, expression1_ ≡ expression2_], {equalities___}] :=
 If[And[
   SimplifyByRewriting[expression1 ≡ expression2 /. (n1 → 0), {equalities}] === Proved ,
   SimplifyByRewriting[expression1 ≡ expression2 /. (n1 → n1⁺),
     {equalities, expression1 :→ expression2}] === Proved],
  Proved, Unproved]
```

Test Examples:

```
ProveByInduction[forAll[{x}, plus[0, x] ≡ x], Axioms𝕅0plus]
```

```
Proved
```

```
ProveByInduction[forAll[{x}, plus[0, x] ≡ 0], Axioms𝕅0plus]
```

```
Unproved
```

```
Clear[SimplifyByRewriting, ProveByInduction]
```

We Want to See the Proof!

For this, we have to **establish a recursive data structure** that keeps track of the “tree” of the individual proof steps (even failing ones) and, at the end, allows us to “read” the entire proof.

Also, the proof should be structured so that we can **“close” whole sub-parts** in the proof if we are not interested in the details.

Roughly, for this, we introduce the following data structure for “reasoning trees” in Mathematica (which is applicable not only for induction proofs and equalities):

```
RT[
 reasoningMethod (* comment on the reasoning method;
 typically, we just use the identifier of the method *),
 goal (* formula to be reasoned on *),
 knowledge (* a list of formulae  *),
 result (*  result of applying the
  reasoningMethod to the goal using the knowledge *),
 {reasoningTree1, reasoningTree2,  ...}  (* list of reasoning subtrees  *)]
```

(Details of the notation will be explained in the lecture. We us “RT” as tag for “reasoning tree”.)

Displaying Reasoning Trees in “Nested Cells Form”

I wrote a little Mathematica program (see the code for ‘NestedCellForm’ in the appendix of the tutorial), which displays nested reasoning trees as Mathematica notebooks with “nested cells” so that entire sub-trees can be closed and opened by clicking at the respective cell brackets.

Example: Our reasoner that also stores the intermediate steps of the proof, for the

```
forAll[{y}, plus[0, y] ≡ y].
```

will produce the folllowing reasoning tree that shows the subproof in subtrees of the nested expression

```
reasoningTree = RT[Prove, forAll[{y}, plus[0, y] ≡ y],
  {plus[x_, 0] → x, plus[x_, y_⁺] → plus[x, y]⁺}, Proved,
   {RT[ProveBySimplificationOrInduction, forAll[{y}, plus[0, y] ≡ y], {}, Proved,
     {RT[ProveBySimplification, plus[0, 0] ≡ 0, {}, Proved,
       {RT[SimplifyByRewriting, plus[0, 0] ≡ 0, {}, 0 ≡ 0]}], RT[ProveBySimplification,
       plus[0, y⁺] ≡ y⁺, {plus[0, 0] :→ 0, plus[0, y] :→ y}, Proved, {RT[SimplifyByRewriting,
         plus[0, y⁺] ≡ y⁺, {plus[0, 0] :→ 0, plus[0, y] :→ y}, y⁺ ≡ y⁺]}]}]}]
```

From there, by our ‘NesteCellsForm’ function we can produce an extra Mathematica notebook that displays the proof in easy-to-read nicely structured form that is, in fact, much nicer to study than a “spaghetti proof” in a text book.

```
reasoningTree // NestedCellsForm
```

```
NotebookObject[ Untitled-37 ]
```

Just try it out.

Introduction

A Crash Course in Mathematica

The Theory of Natural Numbers

Rough Sketch of an Induction Prover

A More Elaborate Induction Prover

Conclusion

Sketch of the Prover

Our induction prover for equalities over the natural numbers with arbitrarily many induction variables has three layers:

`SimplifyByRewriting`

`ProveBySimplification`

`ProveBySimplificationOrInduction`

And a starter

`Prove`

that calls ‘ProveBySimplificationOrInduction’ and displays the initial theory, which is stored in the global variable

`$Theory`

and which is not any more displayed in the subsequent proof steps is for keeping the individual proof cells small.

‘SimplifyByRewriting’

```
Clear[SimplifyByRewriting]

SimplifyByRewriting[expression_, {equalities___}] :=

 RT[SimplifyByRewriting,
  expression,
  {equalities},
  (expression //. ($Theory~Join~{equalities}))]
```

(The program will be explained in detail in the lecture.)

‘ProveBySimplification’

```
Clear[ProveBySimplification]

ProveBySimplification[leftExpression_ ≡ rightExpression_, {equalities___}] :=

 Module[{reasonTree, result, r, lhs, rhs},

  reasonTree = SimplifyByRewriting[leftExpression ≡ rightExpression, {equalities}];
  r = Result[reasonTree]; lhs = r[[1]]; rhs = r[[2]];
  result = If[lhs === rhs, Proved, Unproved];
  (* Result[reasonTree] has the form lhs ≡ rhs,
  where lhs and rhs are simplified w.r.t. the equalities.*)
  RT[ProveBySimplification,
   leftExpression ≡ rightExpression,
   {equalities},
   result,
   {reasonTree}]]
```

(The program will be explained in detail in the lecture.)

‘Prove’

```
Clear[Prove]

Prove[expression_] :=

 Module[{reasonTree, result},

  reasonTree = ProveBySimplificationOrInduction[expression, {}];
  result = Result[reasonTree];
  RT[Prove,
   expression,
   $Theory,
   result,
   {reasonTree}]]
```

(The program will be explained in detail in the lecture.)

‘ProveBySimplificationOrInduction’

```
Clear[ProveBySimplificationOrInduction]

ProveBySimplificationOrInduction[
  forAll[{n1_, n__}, expression_], {equalities___}] :=
 Module[{reasonTree0, reasonTree1, reasonTree2, result},

  reasonTree0 = ProveBySimplification[expression, {equalities}];
  result = Result[reasonTree0];
  If[result === Proved,
   Return[
    RT[ProveBySimplificationOrInduction,
     forAll[{n1, n}, expression], {equalities}, Proved, {reasonTree0}]]];

  reasonTree1 = ProveBySimplificationOrInduction[
    forAll[{n}, expression /. (n1 → 0)], {equalities}];
  result = Result[reasonTree1];
  If[result === Unproved,
   Return[
    RT[ProveBySimplificationOrInduction,
     forAll[{n1, n}, expression], {equalities}, Unproved,
     {reasonTree1}]]];

  reasonTree2 =
   ProveBySimplificationOrInduction[forAll[{n}, expression /. (n1 → n1⁺)],
    {equalities,
     RuleFromEquality[forAll[{n}, expression /. (n1 → 0)]],
     RuleFromEquality[forAll[{n}, expression]]}];
  result = Result[reasonTree2];
  RT[ProveBySimplificationOrInduction,
   forAll[{n1, n}, expression], {equalities}, result,
   {reasonTree1, reasonTree2}]]

ProveBySimplificationOrInduction[forAll[{n1_}, expression_], {equalities___}] :=

 Module[{reasonTree0, reasonTree1, reasonTree2, result},

  reasonTree0 = ProveBySimplification[expression, {equalities}];
  result = Result[reasonTree0];
  If[result === Proved,
   Return[
    RT[ProveBySimplificationOrInduction,
     forAll[{n1}, expression], {equalities}, Proved, {reasonTree0}]]];

  reasonTree1 = ProveBySimplification[expression /. (n1 → 0), {equalities}];
  result = Result[reasonTree1];
  If[result === Unproved,
   Return[
    RT[ProveBySimplificationOrInduction,
     forAll[{n1}, expression], {equalities}, Unproved,
     {reasonTree1}]]];
   reasonTree2 = ProveBySimplification[expression /. (n1 → n1⁺),
     {equalities,
      RuleFromEquality[expression /. (n1 → 0)], RuleFromEquality[expression]}];
   result = Result[reasonTree2];
   RT[ProveBySimplificationOrInduction,
    forAll[{n1}, expression], {equalities}, result,
    {reasonTree1, reasonTree2}]]

ProveBySimplificationOrInduction[expression_, {equalities___}] :=
 ProveBySimplification[expression, {equalities}]
```

(The program will be explained in detail in the lecture.)

Conclusion

I hope I was able to **motivate you**

- to improve your f**ormal proving** skill
- to play with the sample prover to explore **more and more rounds** of (equational inductive) natural number theory
- to expand the prover to parts of natural number theory **beyond equational inductive** theory
- to expand the prover to **other inductive domains**
- to expand the prover to **more general proof methods** and **other mathematical theories**
- to enjoy mathematical theory exploration by **working on the object and the meta level in parallel**
- to use **provers for teaching proving**
- to use the **Theorema system** for formal automated mathematical theory exploration (in case you are interested write to me for obtaining a free license under open source license conditions).

and to give you the feeling that **you are well prepared for the next “revolutions”** in the world of math and software based technology.