Verified Provers

Martin Giese

Radon Institute for Computational and Applied Mathematics

Austrian Academy of Sciences

Linz, Austria

KeY Symposium, Lökeberg, 8 June 2005

Theorema

Theorema aims to be a software system that supports

the *entire process* of mathematical theory exploration:

- invention of concepts
- invention & verification (proof) of propositions about concepts
- invention of problems formulated in terms of concepts
- invention & verification of algorithms to solve problems
- storage and retrieval of all this information

Project started by Bruno Buchberger in 1996/1997

Three Types of Reasoning

Three typical mathematical activities:

- Computing: Given $f(x)$ and a value v for x, compute $f(v)$.

Three Types of Reasoning

Three typical mathematical activities:

- Computing: Given $f(x)$ and a value v for x, compute $f(v)$.
- Solving: Given $P(x)$ find a value v with $\models P(v)$

Three Types of Reasoning

Three typical mathematical activities:

- Computing: Given $f(x)$ and a value v for x, compute $f(v)$.
- Solving: Given $P(x)$ find a value v with $\models P(v)$
- Proving: Show that for all values v, $\models P(v)$

Three Types of Reasoning

Three typical mathematical activities:

- Computing: Given $f(x)$ and a value v for x, compute $f(v)$.
- Solving: Given $P(x)$ find a value v with $\models P(v)$
- Proving: Show that for all values v, $\models P(v)$

Computer algebra systems quite good at Computing and Solving

➠ use these capabilities for Proving.

Mathematical TP vs. Program Verification

KeY wants to make mechanized formal methods

attractive to software people.

Mathematical TP vs. Program Verification

KeY wants to make mechanized formal methods

attractive to software people.

Theorema wants to make mechanized formal methods

attractive to mathematics people.

Mathematical TP vs. Program Verification

KeY wants to make mechanized formal methods

attractive to software people.

Theorema wants to make mechanized formal methods

attractive to mathematics people.

(OK, mathematicians don't like to be too formal either.

But they'll just have to learn)

Syntax

- For software engineers, ASCII is best:

```
==>
 ex x:s. all y:s. p(x, y) -> all v:s. ex u:s. p(u, v)
```

Syntax

- For software engineers, ASCII is best:

```
==>
 ex x:s. all y:s. p(x, y) -> all v:s. ex u:s. p(u, v)
```

- For math, 2D Mathematical Syntax and Symbols are a must

$$\textbf{Definition}\Big[\text{"limit"},\ \underset{f,a}{\forall}\ \text{limit}[f, a] \Longleftrightarrow \underset{\substack{\epsilon \\ \epsilon>0}}{\forall}\ \underset{N}{\exists}\ \underset{\substack{n \\ n\geq N}}{\forall}\ |f[n] - a| < \epsilon \quad \text{"1"}\Big]$$

Syntax

- For software engineers, ASCII is best:

```
==>
 ex x:s. all y:s. p(x, y) -> all v:s. ex u:s. p(u, v)
```

- For math, 2D Mathematical Syntax and Symbols are a must

$$\textbf{Definition}\left[\text{"limit"},\ \underset{f,a}{\forall}\ \text{limit}[f, a] \Longleftrightarrow \underset{\underset{\epsilon>0}{\epsilon}}{\forall}\ \underset{N}{\exists}\ \underset{\underset{n\geq N}{n}}{\forall}\ |f[n] - a| < \epsilon \quad \text{"l"}\right]$$

➠ *Theorema* uses Mathematica front-end.

Logic

- For Software verification, we can decide:
  - Use classical first-order logic (+ program logic)
  - Use type system adapted to the programming language
  - Use a sequent calculus

Logic

- For Software verification, we can decide:
  - Use classical first-order logic (+ program logic)
  - Use type system adapted to the programming language
  - Use a sequent calculus
- Mathematicians
  - might want to use intuitionistic logic, probably higher-order
  - might want to use Martin-Löf's type theory, or Constable's, or Coquand's. Or set theory with no types at all. Or...
  - might require odd calculi

Logic

- For Software verification, we can decide:
  - Use classical first-order logic (+ program logic)
  - Use type system adapted to the programming language
  - Use a sequent calculus
- Mathematicians
  - might want to use intuitionistic logic, probably higher-order
  - might want to use Martin-Löf's type theory, or Constable's, or Coquand's. Or set theory with no types at all. Or...
  - might require odd calculi

➠ Need to keep architecture very open

Domains

- For software:
  - Domains like lists, sets, maps, trees. . . (all finite)
  - Reasoning mostly by induction following data types
  - External 'Simplify' call useful, but not really central

Domains

- For software:
  - Domains like lists, sets, maps, trees. . . (all finite)
  - Reasoning mostly by induction following data types
  - External 'Simplify' call useful, but not really central
- For mathematics:
  - Domains like natural numbers, real numbers, polynomials rings,. . .
  - Various proof techniques, depending on domains
  - *very* powerful decision procedures for some domains (Gröbner bases for algebraic equations, Paule-Schorn method for combinatorial and other special functions, CAD for inequalities)

Extension of Reasoners

- invention of domain-specific simplification or proof methods is part of mathematical exploration
- want to implement these as part of exploration process
- optimally in same mathematical language
- want to formally verify them

Basic ideas:

B. Buchberger: Proving by First and Intermediate Principles

Invited talk at TYPES Workshop, Nov 1-2, 2004

University of Nijmegen, The Netherlands

Taclets?

Couldn't we use taclets and their Proof Obligations for this?

Taclets?

Couldn't we use taclets and their Proof Obligations for this?

Maybe, but

- Taclets are for one logic (classical)

Taclets?

Couldn't we use taclets and their Proof Obligations for this?

Maybe, but

- Taclets are for one logic (classical)
- Taclets are for one calculus (sequents)

Taclets?

Couldn't we use taclets and their Proof Obligations for this?

Maybe, but

- Taclets are for one logic (classical)
- Taclets are for one calculus (sequents)
- Taclets are not for complicated programs

Taclets?

Couldn't we use taclets and their Proof Obligations for this?

Maybe, but

- Taclets are for one logic (classical)
- Taclets are for one calculus (sequents)
- Taclets are not for complicated programs

➠ Probably rather not.

Taclets?

Couldn't we use taclets and their Proof Obligations for this?

Maybe, but

- Taclets are for one logic (classical)
- Taclets are for one calculus (sequents)
- Taclets are not for complicated programs

➠ Probably rather not.

But maybe we can carry some ideas over?

Example

Example: After having shown

$$\forall_{x,y,z} (x+y)+z = x+(y+z)$$

for natural numbers, we decide to simplify all such expressions by shifting parentheses to the right.

For instance

$$((a+b)+(c+d))+e \qquad \rightsquigarrow \qquad a+(b+(c+(d+e)))$$

Data structure

Compute one term from another

➠ need representation for terms.

Restrict to first-order terms:

$$Term = Var[Name] \mid Apply[Name, List[Term]]$$

Example: term $0 + x$ is represented as

$$Apply[\text{"+"}, \{Apply[\text{"0"}, \{\}], Var[\text{"x"}]\}]$$

Color Quoting

Make syntax easier to read by using a colour for quoting:

$$\mathbf{0} + x$$

stands for

$$Apply[\text{“+”}, \{Apply[\text{“0”}, \{\}], Var[\text{“x”}]\}]$$

(x is of type Nat on object level) and

$$\mathbf{0} + x$$

for

$$Apply[\text{“+”}, \{Apply[\text{“0”}, \{\}], x\}]$$

(x is of type $Term$ on meta level)

Implementation of Simplifier

$$shiftParens :: Term \longrightarrow Term$$

$$shiftParens[t] = sumList[collect[t, \{\}]]$$

Implementation of Simplifier

$$shiftParens :: Term \longrightarrow Term$$

$$shiftParens[t] = sumList[collect[t, \{\}]]$$

$$collect :: Term \times List[Term] \longrightarrow List[Term]$$

$$collect[0, acc] = 0.acc$$

$$collect[succ[t], acc] = succ[t].acc$$

$$collect[Var[name], acc] = Var[name].acc$$

$$collect[t_1 + t_2, acc] = collect[t_1, collect[t_2, acc]]$$

Implementation of Simplifier

$$shiftParens :: Term \longrightarrow Term$$

$$shiftParens[t] = sumList[collect[t, \{\}]]$$

$$collect :: Term \times List[Term] \longrightarrow List[Term]$$

$$collect[0, acc] = 0.acc$$

$$collect[succ[t], acc] = succ[t].acc$$

$$collect[Var[name], acc] = Var[name].acc$$

$$collect[t_1 + t_2, acc] = collect[t_1, collect[t_2, acc]]$$

$$sumList :: List[Term] \longrightarrow Term$$

$$sumList[\{\}] = 0$$

$$sumList[\{t\}] = t$$

$$sumList[s.t.ts] = s + sumList[t.ts]$$

Informal Verification

Need to show that the term represented by t and the term represented by $shiftParens[t]$ have the same value.

Informal Verification

Need to show that the term represented by t and the term represented by $shiftParens[t]$ have the same value.

'value' is a model theoretic concept.

➠ not good for (Meta-)mathematics.

➠ prove e.g. that there is a rewriting derivation between the two, using equalities in the knowledge base.

Informal Verification

Need to show that the term represented by t and the term represented by $shiftParens[t]$ have the same value.

'value' is a model theoretic concept.

➠ not good for (Meta-)mathematics.

➠ prove e.g. that there is a rewriting derivation between the two, using equalities in the knowledge base.

Strengthen induction: For any t and l, there is a rewriting between $sumList[collect[t, l]]$ and $t + sumList[l]$.

Informal Verification

Need to show that the term represented by t and the term represented by $shiftParens[t]$ have the same value.

'value' is a model theoretic concept.

➠ not good for (Meta-)mathematics.

➠ prove e.g. that there is a rewriting derivation between the two, using equalities in the knowledge base.

Strengthen induction: For any t and l, there is a rewriting between $sumList[collect[t, l]]$ and $t + sumList[l]$.

. . . (Fairly easy exercise) . . .

Formal Verification

Define predicate: $provablyEq[KB, t_1, t_2]$

Formal Verification

Define predicate: $provablyEq[KB, t_1, t_2]$

Theorems/Axioms include:

$$\bigwedge \begin{cases} l{=}r \in KB \\ subterm[t, pos] = l \end{cases} \quad \Rightarrow \quad provablyEq[KB, t, replace[t, pos, r]]$$

Formal Verification

Define predicate: $provablyEq[KB, t_1, t_2]$

Theorems/Axioms include:

$$\bigwedge \begin{cases} l = r \in KB \\ subterm[t, pos] = l \end{cases} \Rightarrow \quad provablyEq[KB, t, replace[t, pos, r]]$$

$$provablyEq[KB, t, t]$$

Formal Verification

Define predicate: $provablyEq[KB, t_1, t_2]$

Theorems/Axioms include:

$$\bigwedge \left\{ \begin{array}{l} l = r \in KB \\ subterm[t, pos] = l \end{array} \right. \quad \Rightarrow \quad provablyEq[KB, t, replace[t, pos, r]]$$

$$provablyEq[KB, t, t]$$

$$\bigwedge \left\{ \begin{array}{l} provablyEq[KB, t_1, t_2] \\ provablyEq[KB, t_2, t_3] \end{array} \right. \quad \Rightarrow \quad provablyEq[KB, t_1, t_3]$$

Formal Verification

Define predicate: $provablyEq[KB, t_1, t_2]$

Theorems/Axioms include:

$$\bigwedge \begin{cases} l = r \in KB \\ subterm[t, pos] = l \end{cases} \Rightarrow \quad provablyEq[KB, t, replace[t, pos, r]]$$

$$provablyEq[KB, t, t]$$

$$\bigwedge \begin{cases} provablyEq[KB, t_1, t_2] \\ provablyEq[KB, t_2, t_3] \end{cases} \Rightarrow \quad provablyEq[KB, t_1, t_3]$$

Structural induction schema on terms

Formal Verification

Define predicate: $provablyEq[KB, t_1, t_2]$

Theorems/Axioms include:

$$\bigwedge \begin{cases} l = r \in KB \\ subterm[t, pos] = l \end{cases} \Rightarrow \quad provablyEq[KB, t, replace[t, pos, r]]$$

$$provablyEq[KB, t, t]$$

$$\bigwedge \begin{cases} provablyEq[KB, t_1, t_2] \\ provablyEq[KB, t_2, t_3] \end{cases} \Rightarrow \quad provablyEq[KB, t_1, t_3]$$

Structural induction schema on terms

etc., etc. . .

Given some patience, the informal proof should be formalizable.

Plugging it into the Prover

Prover needs to support adding simplifiers:

$$Prove :: KBase \times Term \times List[Term \rightarrow Term] \rightarrow ProofResult$$

Plugging it into the Prover

Prover needs to support adding simplifiers:

$$Prove :: KBase \times Term \times List[Term \longrightarrow Term] \longrightarrow ProofResult$$

Define predicate:

$$isSoundSimplifier(KB, s) :\Leftrightarrow \underset{t \in Term}{\forall} provablyEq[KB, t, s[t]]$$

Plugging it into the Prover

Prover needs to support adding simplifiers:

$$\mathit{Prove} :: \mathit{KBase} \times \mathit{Term} \times \mathit{List}[\mathit{Term} \to \mathit{Term}] \to \mathit{ProofResult}$$

Define predicate:

$$\mathit{isSoundSimplifier}(\mathit{KB}, s) :\Leftrightarrow \underset{t \in \mathit{Term}}{\forall} \mathit{provablyEq}[\mathit{KB}, t, s[t]]$$

Soundness of Prove:

$$\bigwedge \begin{cases} \underset{s \in l}{\forall} \mathit{isSoundSimplifier}[\mathit{KB}, s] \\ \mathit{successful}[\mathit{Prove}[\mathit{KB}, \phi, l]] \end{cases} \Rightarrow \quad \mathit{provable}[\mathit{KB}, \phi]$$

Plugging it into the Prover

Prover needs to support adding simplifiers:

$$Prove :: KBase \times Term \times List[Term \rightarrow Term] \rightarrow ProofResult$$

Define predicate:

$$isSoundSimplifier(KB, s) :\Leftrightarrow \underset{t \in Term}{\forall} provablyEq[KB, t, s[t]]$$

Soundness of $Prove$:

$$\bigwedge \begin{cases} \underset{s \in l}{\forall} isSoundSimplifier[KB, s] \\ successful[Prove[KB, \phi, l]] \end{cases} \quad \Rightarrow \quad provable[KB, \phi]$$

Definition of $provable$, maybe:

$$provable[KB, \phi] :\Leftrightarrow successful[Prove[KB, \phi, \{\}]]$$

Efficient Verification

- Proving everything by induction on terms is tedious.

Efficient Verification

- Proving everything by induction on terms is tedious.
- Recognize common algorithm patterns, e.g. exhaustive application of rewrite rule $(x + y) + z \mapsto x + (y + z)$

Efficient Verification

- Proving everything by induction on terms is tedious.
- Recognize common algorithm patterns, e.g. exhaustive application of rewrite rule $(x + y) + z \mapsto x + (y + z)$
- Capture generic algorithm, e.g. $simplifyByRewriting :: List[RewriteRule] \longrightarrow (Term \longrightarrow Term)$

Efficient Verification

- Proving everything by induction on terms is tedious.
- Recognize common algorithm patterns, e.g. exhaustive application of rewrite rule $(x + y) + z \mapsto x + (y + z)$
- Capture generic algorithm, e.g.
  $simplifyByRewriting :: List[RewriteRule] \longrightarrow (Term \longrightarrow Term)$
- generic soundness:
  $$\forall_{r \in l} isSoundRewriteRule[KB, r] \quad \Rightarrow$$
  $$isSoundSimplifier[KB, simplifyByRewriting[l]]$$

Efficient Verification

- Proving everything by induction on terms is tedious.
- Recognize common algorithm patterns, e.g. exhaustive application of rewrite rule $(x + y) + z \mapsto x + (y + z)$
- Capture generic algorithm, e.g.

  $simplifyByRewriting :: List[RewriteRule] \longrightarrow (Term \longrightarrow Term)$
- generic soundness:

  $$\forall_{r \in l} isSoundRewriteRule[KB, r] \quad \Rightarrow$$
  $$isSoundSimplifier[KB, simplifyByRewriting[l]]$$
- prove this by term induction.

Efficient Verification (cont.)

When is a rewrite rule is sound?

$$isSoundRewriteRule[l \mapsto r] :\Leftrightarrow \underset{\sigma \in Subst}{\forall} provablyEq[\sigma[l], \sigma[r]]$$

Efficient Verification (cont.)

When is a rewrite rule is sound?

$$isSoundRewriteRule[l \mapsto r] :\Leftrightarrow \underset{\sigma \in Subst}{\forall} provablyEq[\sigma[l], \sigma[r]]$$

By the logic's substitution theorem:

$$provable[univClosure[l = r]] \Rightarrow isSoundRewriteRule[l \mapsto r]$$

Efficient Verification (cont.)

When is a rewrite rule is sound?

$$isSoundRewriteRule[l \mapsto r] :\Leftrightarrow \underset{\sigma \in Subst}{\forall} provablyEq[\sigma[l], \sigma[r]]$$

By the logic's substitution theorem:

$$provable[univClosure[l{=}r]] \Rightarrow isSoundRewriteRule[l \mapsto r]$$

E.g. for $(x+y)+z \mapsto x+(y+z)$, show:

$$provable[\underset{x,y,z}{\forall}(x+y)+z = x+(y+z)]$$

Efficient Verification (cont.)

When is a rewrite rule is sound?

$$isSoundRewriteRule[l \mapsto r] :\Leftrightarrow \forall_{\sigma \in Subst} provablyEq[\sigma[l], \sigma[r]]$$

By the logic's substitution theorem:

$$provable[univClosure[l{=}r]] \Rightarrow isSoundRewriteRule[l \mapsto r]$$

E.g. for $(x + y) + z \mapsto x + (y + z)$, show:

$$provable[\forall_{x,y,z} (x + y) + z = x + (y + z)]$$

Which is the same as proving

$$\forall_{x,y,z} (x + y) + z = x + (y + z)$$

Efficient Verification (cont.)

When is a rewrite rule is sound?

$$isSoundRewriteRule[l \mapsto r] :\Leftrightarrow \underset{\sigma \in Subst}{\forall} provablyEq[\sigma[l], \sigma[r]]$$

By the logic's substitution theorem:

$$provable[univClosure[l = r]] \Rightarrow isSoundRewriteRule[l \mapsto r]$$

E.g. for $(x + y) + z \mapsto x + (y + z)$, show:

$$provable[\underset{x,y,z}{\forall}(x + y) + z = x + (y + z)]$$

Which is the same as proving

$$\underset{x,y,z}{\forall}(x + y) + z = x + (y + z)$$

on the object level!

Conclusion

- *Theorema* is a system for mathematical theory exploration.
- Mathematical theorem proving is different from TP for verification.
- Extension of reasoners requires full program verification in general.
- Maybe often possible to use object level reasoning like for taclets.