Theorems and Algorithms: An Interface between Isabelle and Maple

Clemens Ballarin   Karsten Homann   Jacques Calmet

Universität Karlsruhe
Institut für Algorithmen und Kognitive Systeme
Am Fasanengarten 5 · 76131 Karlsruhe · Germany
{ballarin, homann, calmet}@ira.uka.de

Abstract

Solving sophisticated mathematical problems often requires algebraic algorithms *and* theorems. However, there are no environments integrating theorem provers and computer algebra systems which consistently provide the inference capabilities of the first and the powerful arithmetic of the latter systems.

As an example for such a mechanized mathematics environment we describe a prototype implementation of an interface between Isabelle and Maple. It is achieved by extending the simplifier of Isabelle through the introduction of a new class of simplification rules called evaluation rules in order to make selected operations of Maple available, and without any modification to the computer algebra system. Additionaly, we specify syntax translations for the concrete syntax of Maple which enables the communication between both systems illustrated by some examples that can be solved by theorems and algorithms.

1 Introduction

Problem solving in mathematics often requires the application of both procedural algebraic knowledge (algorithms) and deductive knowledge (theorems). The advantages of combining both strategies have been recognized by both communities: symbolic computation and analytical reasoning. Some of these advantages concern the introduction of mathematical theories and arithmetics, in particular real numbers, into provers, as well as providing logical languages and justifications to symbolic calculators. Two aspects must be further investigated: (i) the problem of combining algorithms and theorems in one single system, (ii) the heterogeneous integration of several packages.

On the one hand classical computer algebra systems (CAS), for example Maple [Char et al. 92] or Mathematica [Wolfram 91], usually offer a straightforward programming language with ad-hoc implementations of rewriting. One of the most promising approaches towards introducing theorem proving into CAS is an extension of Analytica [Clarke & Zhao 94], a Mathematica package to prove theorems in elementary analysis. It is able to deal with the internal mathematical knowledge of Mathematica and guarantees the correctness of certain operations, e.g. prevents division by zero. The package can solve an extensive collection of nontrivial mathematical problems.

On the other hand theorem proving has shown to be an important field interfacing artificial intelligence and mathematics. It attempts to perform symbolic calculations of mathematical proofs by computers. Some classical theorem provers (TP) were extended by techniques of symbolic computing, e.g. Otter [McCune 94] allows to call external algorithms out of proofs. Specialized prover packages have been developed which are capable of performing symbolic mathematical computations. For example, the interactive theorem proving project [Ueberberg 94] developed a program for the use of CAS to support mathematicians in proving theorems in incidence geometry. However, there are no environments integrating theorem provers and computer algebra systems which consistently provide the inference capabilities of the first and the powerful arithmetic of the latter systems.

Another aspect is the integration of several systems in a common environment. Different possibilities to integrate symbolic calculators and theorem provers are given in [Homann & Calmet 94]. Since it is doubtful that one single system can satisfy the multitude and divergency of requests and problems the goal of this research is to a certain extent heterogeneous in the sense of integrating diverse systems. Ideally, such an *open mechanized mathematics environment* should be easily extensible and should provide interfaces to the existing widespread CAS and TP.

Some works exist to integrate different TP or CAS respectively. [Giunchiglia et al. 94] introduces an architecture for open mechanized reasoning systems which consists of a reasoning theory as well as a control and an interaction component. The long term goal is a methodology to construct complex systems as a composition of several reasoning systems. CAS/π [Kajler 92] is a powerful system-independent graphical user interface to some CAS (Maple, Sisyphe, Ulysse). It was designed so that expert users can set up connections to alternative CAS easily and at runtime.

[Harrison & Théry 93] describes a bridge between the theorem prover HOL and CAS. The CAS are used as an oracle guiding the proof which is still done rigorously in HOL. In accordance to [Homann & Calmet 94] this interaction is classified as a master-slave relation (HOL as master), without a common knowledge representation, and with no trust at all. The advantage of this architecture is that certain algebraic operations become available inside the prover without

In A.H.M. Levelt (Ed.),
Proceedings of International Symposium on Symbolic and Algebraic Computation, **ISSAC'95**,
Montréal, Canada, ACM Press, pp. 150–157, 1995

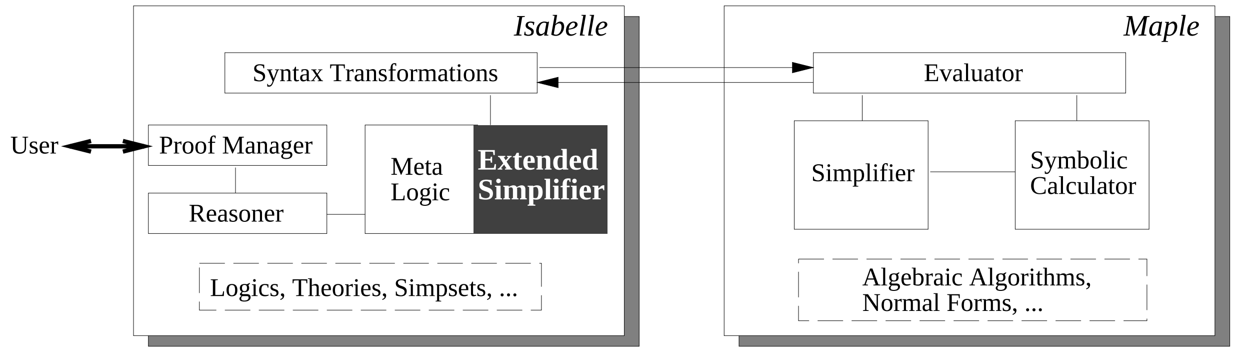

Figure 1: Schematic Link between Isabelle and Maple

jeopardizing the security of the results. However, the prover must be able to verify the results of the symbolic calculator. Another approach is given in [Jackson 94] and presents an interaction between Nuprl and the Weyl computer algebra system. In this case, Nuprl provides abstract algebra in expressive type theory and acts as an algebraic oracle to Weyl. Again, both systems behave like black boxes, Weyl as master, without common representation, but trusting Nuprl.

The integration presented in this paper describes a prototype implementation of an interface with Maple [Char et al. 92] as a slave to the tactical theorem prover Isabelle [Paulson 94]. It is a first step towards the development of an open heterogeneous environment for doing mathematics. The interface is designed by extending the simplifier of Isabelle without any modification of Maple. Since we do not have to take into consideration any idiosyncrasies of Maple except its syntax, it is very easy to link Isabelle to other CAS as well. Unfortunately, this approach does not allow to easily exchange Isabelle by another theorem prover. We discuss a more general environment in [Calmet & Homann 95].

The simplifier of Isabelle is extended by introducing a new class of simplification rules called evaluation rules in order to make selected operations of Maple available. Additionaly, we specify syntax translations for the concrete syntax of Maple. They enable Isabelle to communicate with the computer algebra system as illustrated in figure 1.

This paper is organized as follows. Section 2 introduces the basic concepts of Isabelle, i.e. theories and proofs, as deep as required to describe the interface to Maple and its semantics. Section 3 presents this interface as an extension of Isabelle's simplifier. Section 4 describes how to link Maple to Isabelle by specifying syntax translations and defining evaluation rules. Section 5 presents some examples. Although elementary, they offer good insight into the problems which can be solved by theorems *and* algorithms. In the last section, we summarize the results and discuss our future work.

2 The tactical theorem prover Isabelle

We present an interface with Maple as a slave to the tactical prover Isabelle acting as master and controlling the proof search. The concept of a tactical theorem prover was introduced by Robin Milner [Milner 85]. Theorems are modeled as an abstract data type with deduction rules as operations. Proofs are programs in the meta-language (usually ML). The advantage of this approach is that the user has the spectrum from doing manual proof steps to applying complex search strategies. This section introduces Isabelle only as far as needed for our work. A more detailed introduction to ML and Isabelle is given in [Milner & Tofte 91] and [Paulson 94] respectively.

Isabelle implements a meta-logic. This is based upon simply typed λ-calculus and has the connectives $\Longrightarrow$ which expresses entailment, the meta-quantifier $\bigwedge$ which expresses generality in rules and axiom schemes, and the equality $\equiv$. The meta-logic also contains axioms like $P \Longrightarrow P$ and deduction rules.

2.1 Theories

The meta-logic is used to specify the object-logic. The object-logic allows to model the domain to reason in. Fortunately, one need not build one's own object-logic from scratch, but may extend one of the various already existing logics for Isabelle. Object-logics are specified using *theory definitions* (figure 2).

```
<theory> = <parent theories>
         + <type definitions>
         + <constant symbols>
         + <syntax translations>
         + <rules>
```

Figure 2: Structure of an Isabelle theory definition

Note that functions and quantifiers are constants of a function type and therefore can be declared in the constants' section. Syntax translations play an important role in our implementation, because they are used to specify the concrete syntax of Maple. This syntax is taken over into Isabelle and thus Isabelle's parsing and pretty printing functions are used for the communication with the computer algebra system. Axioms are specified in the rules' section of a theory.

2.2 Construction of backward proofs

Isabelle's main deduction rule is the resolution rule

$$\frac{[\![\psi_1;\ldots;\psi_{\mathrm{m}}]\!] \Longrightarrow \psi \qquad [\![\phi_1;\ldots;\phi_{\mathrm{n}}]\!] \Longrightarrow \phi}{([\![\phi_1;\ldots;\phi_{\mathrm{i-1}};\psi_1;\ldots;\psi_{\mathrm{m}};\phi_{\mathrm{i+1}};\ldots;\phi_{\mathrm{n}}]\!] \Longrightarrow \phi)\mathrm{s}}(\psi s \equiv \phi_i s).$$

In backward proofs a theorem $[\![\phi_1;\ldots;\phi_{\mathrm{n}}]\!] \Longrightarrow \phi$ is viewed as a *proof state* with *main goal* ϕ and *subgoals* $\phi_1,\ldots,\phi_n$. A proof of a formula ϕ starts with the initial proof state $\phi \Longrightarrow \phi$.

In order to refine the ith subgoal with a rule the following resolution on the proof state is performed:

$$\frac{\text{rule} \qquad \text{proof state}}{\text{new proof state}}$$

The head of the rule is unified with the ith subgoal[1]. Functions that operate on the proof states are called tactics. Isabelle also provides functions that combine tactics, called tacticals. They can implement search strategies like depth first search.

The proof state is managed by Isabelle's subgoal module. The following commands are used to control a proof[2]:

```
goal   : theory -> string -> thm list
by     : tactic -> unit
result : unit -> thm
```

`goal` *theory formula*; starts a new proof. *theory* identifies the theory. The goal is given as a ML-string. `goal` returns the premises of *formula*, which are needed in the proof.

`by` *tactic*; applies the tactic *tactic* to the proof state.

`result`(); returns the proved theorem. The proof state must therefore pass certain correctness tests.

The most important tactics are:

```
resolve_tac  : thm list -> int -> tactic
res_inst_tac : (string * string) list -> thm
               -> int -> tactic
assume_tac   : int -> tactic
```

`resolve_tac` *thms i*; refines the proof state using the object-rules *thms*. Resolution is applied to the ith subgoal with the first matching rule.

`res_inst_tac` *insts thm i*; this resolution tactic is used to apply rules like substitution and induction. Such rules usually allow a huge number of unifiers. The tactic takes a list of explicit instantiations of variables in the rule.

`assume_tac` *i*; attempts to solve subgoal i by assumption.

3 The Interface

3.1 Computer algebra systems as term rewrite systems

Computer algebra systems contain equation solvers. It would be desirable that they could be used as unification procedures for algebraic equational theories. Unfortunately, even polynomial expressions over integers that contain variables cannot be unified. This is a simple consequence of Matiyasevič's theorem stating that Hilbert's tenth problem is unsolvable.

We consider the computer algebra system as a term rewrite system. As shown in figure 3 conditional rewrite rules can be used to model all the calculations that are performed by computer algebra systems[3].

Premises in the rules are important to prevent the computer algebra system from incorrect calculations. In the design of the hybrid system these premises have to be selected carefully. Consider the following Maple session.

- Reduction to normal forms:
  $(X^2 - X\cdot 3)(a + X\cdot X) \quad \longrightarrow \quad X^4 - 3X^3 + aX^2 - 3aX$
- Execution of algorithms:
  $\text{solutions}(\begin{pmatrix} 4 & 6 \\ 2 & 3 \end{pmatrix} x = 0, x) \quad \longrightarrow \quad [\begin{pmatrix} 3 \\ -2 \end{pmatrix}]$
- Conditional rewrite rules express premises and domain restrictions:
  $a \neq 0 \quad \Rightarrow \quad \frac{a}{a} \longrightarrow 1$

Figure 3: A computer algebra system as a rewrite system

```
> a = 0;
                    a = 0
> "/a;
                    1 = 0
```

The equation $1 = 0$ is derived, which will lead to a contradiction when assuming that a field is the underlying domain!

3.2 Semantics of the hybrid system

The action of the computer algebra system is embedded into the prover by the schematic formula

$$[\![\mathrm{P}_1, \ldots, \mathrm{P}_n]\!] \Longrightarrow \mathrm{t} \equiv \mathrm{t}'.$$

Provided that the premises $P_1, \ldots, P_n$ hold, the term t is transmitted to the computer algebra system and reduced yielding t'. The computer algebra system may simplify only distinct terms. Depending on the application it should not (or *cannot*) reduce terms that contain logical connectives like $\Longrightarrow, \wedge, \vee$ etc. Isabelle's many-sorted term system allows to treat this conveniently. The notion *consistency w.r.t. signature* is defined. Let Σ be the signature of the object-logic and $A \subseteq \Sigma$ the subsignature that can be understood by the computer algebra system.

Definition 1 *The term algebra $T_\Sigma(X)$ is called* consistent w.r.t. the signature *A, iff for all constant symbols $f \in A$ any term $f(a_1, \ldots, a_n) \in T_\Sigma(X)$ already lies in $T_A(X)$.*

In a consistent term algebra all subterms of a term are in the subsignature $T_A(X)$, provided that the outermost connective belongs to the subsignature A. It is thus easy to recognize terms that can be passed to the computer algebra system during simplification. It only has to be verified that the outermost connective lies in A.

Closure w.r.t. signatures ensures that a term t' returned by the computer algebra system (viewed as an operator E) does not contain unknown function symbols. This notion is necessary because algorithms in computer algebra systems are very powerful and thus may produce "unexpected" results, for instance use expressions that lie in larger domains to express the result, as the following summation example from Maple shows:

```
> sum( k/(k+1), k=0..n );
         n + 1 - Psi(n + 2) - gamma
```

The result is a rational number but the application of the polygamma function ψ and Euler's constant γ are not.

The following definition ensures that the results returned by the computer algebra system are "meaningful" in the sense that their signature lies in the object-logics signature.

[1]If the subgoal contains assumptions or bound variables, the rule is lifted first. Details are given in [Paulson 94].

[2]The colon is followed by the ML-type. `->` denotes a function.

[3]This approach does not permit to declare new functions during a proof session. But this is not a real restriction: functions have to be declared while a theory is designed.

Definition 2 *Let $A \subseteq B \subseteq \Sigma$. Then the operator $E : T_A(X) \rightarrow T_B(X)$ is* closed w.r.t. A and B.

It is straightforward to give the semantics of an object-logic that is extended by a link to a computer algebra system. The semantics of an ordinary object-logic is defined by its axioms and by the meta-logic. With an *enriched* object-logic new axioms may be generated at run time from the *evaluation rules* and the computer algebra system's result. The evaluation rules may increase the logical power of an object-logic. In order to obtain the semantics for every *evaluation rule* all axioms that can be generated that way must be added. Let A be the signature of terms that may be passed to the computer algebra system, B the signature of the terms that may be returned by the system, Σ the signature of the object-logic, $A \subseteq B \subseteq \Sigma$, $T_\Sigma(X)$ *consistent w.r.t. the signature* A, and E the evaluation function of the computer algebra system that is closed w.r.t. A and B. For every *evaluation rule* with premises $P_1, \ldots, P_n$ that can be applied to a term $t \in T_A(X)$ the axiom

$$[\![\mathrm{P}_1, \ldots, \mathrm{P}_n]\!] \Longrightarrow \mathrm{t} \equiv \mathrm{E}(\mathrm{t})$$

has to be added to the object-logic.

The model of an enriched object-logic is obtained by these (infinitely many) axioms. The designer of an enriched object-logic has to make sure that the specified axioms and the evaluation rules are not inconsistent.

3.3 Extension of Isabelle's simplifier

Isabelle's generic simplifier can be set up for object-logics. It works for the equality relation $\equiv$, performs unconditional and conditional rewriting, and uses contextual information ("local assumptions"). Isabelle's simplification tactics are controlled by *simpsets*. These contain a collection of rewrite rules. The simplifier automatically tries to solve conditions that arise from conditional rules by recursive application of the rewrite rules.

We extend this simplifier and introduce a new class of simplification rules that control access to the computer algebra system. These rules are the *evaluation rules* described in the previous paragraph. They are represented as data-structures which contain a list of premises, a term pattern, and the name of a function which enables to call in the computer algebra system. For example, the following evaluation rules could be specified for the natural numbers:

premises	term	call of Maple
—	$a + b$	*eval_map*
$b \leq a$	$a - b$	*eval_map*
—	$a * b$	*eval_map*
$b \mid a, \quad b \neq 0$	a/b	*eval_map*

Isabelle's rewriting system is based on *conversions*[4]. These are functions that take a (meta-)simpset and a term as arguments and return a simplified term. The basic conversion is *rewritec prover*. It applies one rewrite rule and tries to solve conditions arising from the application of a conditional rule using *prover*. `subc` recursively simplifies all the subterms of a term. We introduce a new conversion `evalc` that behaves like `rewritec` but applies an evaluation rule rather than a rewrite rule to the term. Another difference is that `evalc` tries to prove the premises of conditional rules *before* it applies them. This prevents illegal calls of the computer algebra system, e.g. division by zero. Our simplification procedure is outlined by the following pseudo-code.

```
fun botc mss t =
  let val t1 = subc mss t
  in case evalc prover mss t1 of
      None => (case rewritec prover mss t1 of
        None => t1
      | Some(t2) => botc mss t2)
    | Some(t2) => t2
  end
```

After recursively rewriting all the subterms, evaluation rules are tried first. If they are not successful, rewrite rules are tried. Otherwise the result is returned. Contrary to the application of an evaluation rule, simplification is started again after the application of a rewrite rule. It is assumed that the computer algebra system has already tried all possible reductions. This procedure can be refined further, e.g. after trying to find maximal subterms the computer algebra system can process them in one step without simplifying all smaller subterms.

An appropriately chosen prover can derive premises of evaluation rules automatically. This makes conveniently access to the computer algebra system.

3.4 Interface of the extended simplifier

The signature of the extended simplifier is shown in figure 4. Simpsets can be constructed using the operations `empty_ss`, `addsimps`, `addevals` and `addeqcongs`. `setsubgoaler` selects the prover, i.e. an arbitrary tactic.

Simplification is applied to the proof state using the tactics `simp_tac`, `asm_simp_tac` and `asm_full_simp_tac`. All three tactics expect a simpset and a subgoal number. `asm_simp_tac` uses assumptions as additional rewrite rules and `asm_full_simp_tac` simplifies these assumptions one by one, using each assumption in the simplification of the following ones.

4 Linking Maple to Isabelle

Our system is a prototype implementation. As mentioned previously, we are developing a heterogeneous environment which enables to integrate many theorem provers and computer algebra systems. We could have chosen another computer algebra system instead of Maple.

Three simple functions make Maple available within Isabelle's ML-environment.

```
start_maple : string -> unit
exit_maple  : unit -> unit
eval_maple  : string -> string
```

`start_maple` *name*; starts a Maple session (the pretty-printing mode being turned off) and links it to the ML-environment Isabelle is running in using simple Unix-pipes. This function also passes the file *name* to the computer algebra system. The file may contain initialising Maple code, e.g. definitions of new procedures.

`exit_maple ()`; exits the Maple session.

[4]The following is a somewhat simplified description of Isabelle's internals, but it is a good framework to explain what we did. The mechanism of conversions is explained in detail in [Paulson 83].

```
infix addsimps addeqcongs addevals delsimps
      setsolver setloop setmksimps setsubgoaler;

signature E_SIMPLIFIER =
sig
  type simpset
  val simp_tac:          simpset -> int -> tactic
  val asm_simp_tac:      simpset -> int -> tactic
  val asm_full_simp_tac: simpset -> int -> tactic
  val addeqcongs:        simpset * thm list -> simpset
  val addsimps:          simpset * thm list -> simpset
  val delsimps:          simpset * thm list -> simpset
  val addevals:          simpset * (Sign.sg * string list * string * string *
                                     (thm -> term -> thm)) list -> simpset
  val empty_ss:          simpset
  val merge_ss:          simpset * simpset -> simpset
  val strip_evals_ss:    simpset -> simpset
  val setsolver:         simpset * (thm list -> int -> tactic) -> simpset
  val setloop:           simpset * (int -> tactic) -> simpset
  val setmksimps:        simpset * (thm -> thm list) -> simpset
  val setsubgoaler:      simpset * (simpset -> int -> tactic) ->  simpset
  val prems_of_ss:       simpset -> thm list
  val rep_ss:            simpset -> {simps: thm list, congs: thm list}
  val rep_evals:         simpset -> (Sign.sg * string list * string * string *
                                      (Sign.sg -> term -> term)) list
end;
```

Figure 4: Signature of the extended simplifier

`eval_maple` *term*; passes *term* to Maple. Semicolon and linefeed are added to start evaluation. Maple's result is returned as a string. Very large expressions that are cut into several lines by Maple are reconstructed by this function and returned as a single string.

To integrate Maple within an Isabelle theory two steps need to be done: specifying the concrete syntax of Maple and defining evaluation rules in order to make selected operations of Maple available.

4.1 Specifying the concrete syntax

Functions and operations are specified in the constants' section of an Isabelle theory. Functions need a function type as illustrated by the following two examples:

```
igcd   :: "[a, a] => a"
expand :: "a => a"
```

We define `a` to be the type of all objects the computer algebra system deals with. Infix operations are supported by Isabelle. Syntax translations are generated automatically. One has to choose precedence numbers that reflect Maple's precedences, e.g.:

```
"*"    :: "[a, a] => a"         (infixl 70)
"<"    :: "[a, a] => bool"      (infixl 50)
```

The syntax of Maple's higher order functions like sum(f, i=k..l) can also be specified in Isabelle. In Isabelle's type theoretic framework this quantifier is declared

```
FinSum :: "[a => a, a, a] => a"
```

as internal representation. The arguments of `FinSum` are a function and the two boundaries of the summation. In comparison Maple's syntax specifies an expression and the bound variable. This is

```
"@FinSum" :: "[a, idt, a, a] => a"
                 ("(4sum'(_, _=_.._'))" 100)
```

where the string between parentheses specifies Maple's mixfix-syntax. The translation

```
"sum(f, j=k..l)" == "FinSum(%j.f, k, l)"
```

converts between the two representations automatically while printing and parsing. `%j.f` is the λ-notation of the function with argument `j` and body `f`.

4.2 The extended simpsets

It is not possible to model domains in Isabelle's type system, because it does not handle subtyping information like `Natural` $\subseteq$ `Integer`. Therefore, we model domains as sets and, for example, define `Nat` to be the natural numbers. We define the simpset `Nat_ss`. This contains rewrite rules providing typechecking information, and evaluation rules. The latter specify that all terms with a connective of the natural numbers as outermost function symbol can be passed to Maple.

Typechecking rules are axioms of our theory. Some examples are (the colon is Isabelle's symbol for the membership relation):

```
add_type "[| a: Nat; b: Nat |] ==> a + b : Nat"
sub_type "[| a: Nat; b: Nat;
             b <= a |]         ==> a - b : Nat"
```

```
sum_type "[| k: Nat; l: Nat;
           ALL i:Nat. k<=i & i<=l --> f(i): Nat |]
          ==> FinSum(f, k, l) : Nat"
```

Evaluation rules are written as quintuples

$$(sign,\ ["P_1",\ldots,\ "P_n"],\ "t",\ "tT",\ eval)$$

where *sign* gives information for printing and parsing. The second expression is the list of premises, the third is the term to be matched, the fourth its type and the fifth the function to be called.

For the above operations we have the corresponding evaluation rules:

```
  (sign_of Decimal.thy,
   ["a: Nat", "b: Nat"], "a + b", "a", eval_map)
  (sign_of Decimal.thy,
   ["a: Nat", "b: Nat", "b::a <= a"], "a - b",
   "a", eval_map)
  (sign_of Decimal.thy,
   ["k: Nat", "l: Nat",
    "ALL i:Nat. k<=i & i<=l --> f(i): Nat"],
   "FinSum(f, k, l)", "a", eval_map)
```

The type of the expression to be matched is always a. The evaluation function eval_map is composed of the function eval_maple and Isabelle's functions for printing and parsing.

The interaction between typechecking rules and evaluation rules during simplification is illustrated by some examples in the next section.

5 Examples

The examples presented in this section use the computer algebra component of our system for calculations on natural numbers. They can be proved by Isabelle alone using Peano arithmetic, but this approach has linear space complexity for the representation of numbers, and is very inefficient if "big" numbers occur. The benefit of calling Maple is its efficient arithmetics.

5.1 Calculations with natural numbers

In this section we verify the equation $1+9\times 11 = 100$, trivial to a human but not to a theorem prover! The internal representation of numbers are lists of digits. One very inefficient possibility is to give rewrite rules that perform addition and multiplication on such lists. Paulson demonstrated this for binary numbers and managed to derive the product out of two ten-bit numbers within 14 seconds [Paulson 94]!

Our proof starts with the goal command. (Input lines are identified by a dash.)

```
- goal Decimal.thy "1 + 9 * 11 = 100";
Level 0
1 + 9 * 11 = 100
 1. 1 + 9 * 11 = 100
val it = [] : thm list
```

Isabelle returns the proof level as well as the main goal and all subgoals. At the beginning the main goal equals the only subgoal. We start to solve it by simplification:

```
- by (simp_tac Nat_ss 1);
Checking premises for evaluation of:
[| 9 : Nat; 11 : Nat |] ==> 9 * 11 == ?CompAlgSys
  Rewriting:
  9 : Nat == True
  Rewriting:
  11 : Nat == True
SUCCEEDED
9 * 11 == 99
Checking premises for evaluation of:
[| 1 : Nat; 99 : Nat |] ==> 1 + 99 == ?CompAlgSys
  Rewriting:
  1 : Nat == True
  Rewriting:
  99 : Nat == True
SUCCEEDED
1 + 99 == 100
Checking premises for evaluation of:
[| 100 : Nat; 100 : Nat |] ==> 100 = 100 ==
?CompAlgSys
  Rewriting:
  100 : Nat == True
  Rewriting:
  100 : Nat == True
SUCCEEDED
100 = 100 == True
Level 1
1 + 9 * 11 = 100
No subgoals!
val it = () : unit
```

In order to illustrate the intermediate steps and the interaction between rewrite rules for typechecking and evaluation rules, the trace mode of the simplifier was turned on. In this proof Maple performs the rewrites $9 \times 11 \longrightarrow 99$ and $1 + 99 \longrightarrow 100$. Finally, it verifies the equation $100 = 100$, trivial for Isabelle too. One can get the result and bind it to an identifier.

```
- val test1 = result();
val test1 = "1 + 9 * 11 = 100" : thm
```

5.2 A finite summation

Maple contains a collection of algorithms to handle finite summation. Let us consider the following equation

$$\sum_{n=0}^{k}(2n+1) = (k+1)^2.$$

Although this can be proved in our system without an induction, the proof is still cumbersome because we have to establish the premise of the evaluation rule for summation, stating that all items summed up are natural numbers. We omit the three step proof that does not involve the computer algebra system and give only the result as a lemma:

```
val lemma = "?k : Nat ==> ALL i:Nat.
  0 <= i & i <= ?k --> 2 * i + 1 : Nat" : thm
```

The proof is straightforward:

```
- val [prem] = goal Decimal.thy "k: Nat ==>
= sum(2*n+1, n=0..k) = (k+1)^2";
Level 0
sum(2 * n + 1, n=0..k) = (k + 1) ^ 2
 1. sum(2 * n + 1, n=0..k) = (k + 1) ^ 2
val prem = "k : Nat  [k : Nat]" : thm
```

The premise is bound to the identifier prem and used with lemma as a rewrite rule in the proof.

```
- by (simp_tac (Nat_ss addsimps [prem, lemma]) 1);
Level 1
sum(2 * n + 1, n=0..k) = (k + 1) ^ 2
No subgoals!
val it = () : unit
- val test3 = result();
val test3 = "?k : Nat ==> sum(2 * n + 1, n=0..?k)
  = (?k + 1) ^ 2" : thm
```

Isabelle has replaced the variable k by the schematic variable $?k$. This makes it possible to use the theorem with any instantiation for k.

5.3 An induction proof

Our last example gives a formal proof for

$$[\![\mathrm{n} : \mathrm{Nat}; 5 <= \mathrm{n}]\!] \Longrightarrow \mathrm{n}^5 \leq 5^{\mathrm{n}}.$$

The proof uses the induction rule

```
induct
"[| n: Nat; a: Nat;
    P(a);
    !!x. [| x: Nat; a <= x; P(x) |] ==> P(x + 1);
    a <= n |] ==> P(n)"
```

and the two congruence properties for the $\leq$-relation:

```
le_add  "[| a: Nat; b: Nat; c: Nat; d: Nat;
            a <= b; c <= d |] ==> a + c <= b + d"
le_mult "[| a: Nat; b: Nat; c: Nat; d: Nat;
            a <= b; c <= d |] ==> a * c <= b * d"
```

Starting the proof and applying the induction rule leads to:

```
- val [n_nat, bound] = goal Decimal.thy
= " [| n: Nat; 5 <= n |] ==> n ^ 5 <= 5 ^ n";
Level 0
n ^ 5 <= 5 ^ n
 1. n ^ 5 <= 5 ^ n
val n_nat = "n : Nat  [n : Nat]" : thm
val bound = "5 <= n  [5 <= n]" : thm
- TYPE_Nat_chks := Nat_typechecks @ [n_nat];
val it = () : unit
- by (TYPE_Nat (res_inst_tac
= [("a", "5"), ("n", "n")] Decimal.induct 1));
Level 1
n ^ 5 <= 5 ^ n
 1. 5 ^ 5 <= 5 ^ 5
 2. !!x. [| x : Nat; 5 <= x; x ^ 5 <= 5 ^ x |] ==>
         (x + 1) ^ 5 <= 5 ^ (x + 1)
 3. 5 <= n
val it = () : unit
```

a was explicitly instantiated to 5 and the induction variable is n. TYPE_Nat is a tactical we designed to solve subgoals that provide typing information. In this case the first two subgoals correspond to the first premises of the induction rule. The first subgoal corresponds to the base of the induction, the second to the induction step and the third states that the theorem is only valid for n greater or equal to the base case. The following two steps of the proof solve this boundary condition and the induction basis by the assumption bound and the reflexivity of $\leq$ respectively.

```
- by (rtac bound 3);
Level 2
n ^ 5 <= 5 ^ n
 1. 5 ^ 5 <= 5 ^ 5
 2. !!x. [| x : Nat; 5 <= x; x ^ 5 <= 5 ^ x |] ==>
         (x + 1) ^ 5 <= 5 ^ (x + 1)
val it = () : unit
- by (rtac le_refl 1);
Level 3
n ^ 5 <= 5 ^ n
 1. !!x. [| x : Nat; 5 <= x; x ^ 5 <= 5 ^ x |] ==>
    (x + 1) ^ 5 <= 5 ^ (x + 1)
val it = () : unit
```

To expand both sides of the conclusion we introduce the function symbol expand twice before Maple is called to expand the products.

```
- by (res_inst_tac
= [("P", "%y.y <= 5 ^ (x + 1)")] expandE 1);
...
- by (res_inst_tac [("P",
= "%y.expand((x + 1) ^ 5) <= y")] expandE 1);
Level 5
n ^ 5 <= 5 ^ n
 1. !!x. [| x : Nat; 5 <= x; x ^ 5 <= 5 ^ x |] ==>
        expand((x + 1) ^ 5) <= expand(5 ^ (x + 1))
val it = () : unit
- by (asm_simp_tac Nat_simplify_ss 1);
Level 6
n ^ 5 <= 5 ^ n
 1. !!x. [| x : Nat; 5 <= x; x ^ 5 <= 5 ^ x |] ==>
         x ^ 5 + 5 * x ^ 4 + 10 * x ^ 3 +
         10 * x ^ 2 + 5 * x + 1 <= 5 * 5 ^ x
val it = () : unit
```

The following estimates, e.g. $5 * x^4 \leq 5^x$, are proved using the assumption bound, the induction hypothesis, and the rules le_add and le_mult. Our manual proof takes many additional steps to eliminate all subgoals. It can be found in [BALLARIN 94]. A suitable search strategy could do this automatically, for example by setting up Isabelle's original simplifier to use the extended simplifier.

6 Conclusion

We have outlined a methodology for linking algebraic algorithms whose properties can be specified as axioms to the theorem prover Isabelle. This enables the prover to access these properties. Isabelle supports links to external systems through easy formalization of the required "theories" in any syntax, in this case the syntax of Maple. Additionally, Isabelle supports choosing granularity to automate the search for complex proofs. The interface was designed as part of an extended simplifier of the prover such that no additional extensions of Isabelle's kernel are required to link to another computer algebra system.

Integrating algorithms in theorem proving is particularly useful when handling complex expressions or numbers, especially in mathematics. The mathematical knowledge of computer algebra systems offer a powerful tool, e.g. term orders for associative and commutative equational theories. Moreover, the given examples illustrate how CAS can be used at the heuristic level of proof search.

The development of this interface is a prototype and testbed for future efforts. It is part of our research project $\lambda\epsilon\mu\mu\alpha$[5] for problem solving in mathematics based upon

[5]Learning Environment for Mathematics and Mathematical Applications

CAS, TP, knowledge representation systems, explanation-based learning, and distributed AI. Beside the implementation of mathematical "theories" further directions must be investigated:

- Type systems play a crucial part in modern computer algebra systems which are more than only extensive collections of algebraic algorithms. They are of particular importance in symbolic mathematical computing as they can contribute to guarantee correctness of computations and proofs. Since no other CAS has a powerful type system, the integration of Axiom appears to be promising. The developers of Isabelle have been working on providing type classes to their prover as well. However, the concept of subtyping is disregarded while being important to CAS.

- We classified different types of architectures to combine theorem proving and symbolic mathematical computing [HOMANN & CALMET 94]. The prototype presented in this paper corresponds to an architecture consisting of a computer algebra system, a theorem prover, and a common evaluator (bridge). The best possible mutual benefit is achieved either by a common knowledge representation or by standardized interfaces between all systems for doing mathematics. The former requires redesign of present CAS and TP or development of new systems. Such a development may lead to a common and explicit representation of the embedded mathematical knowledge, e.g. theorems, algorithms and types. Moreover, representing theorems corresponding to algebraic algorithms explicitly allows the systems to reflect and explain their behaviour instead of being black boxes. The design of the latter demands to develop a common and general communication language and appropriate interfaces. The capability to connect such systems in a straightforward manner is the main step towards an open mechanized mathematics environment.

Acknowledgement

We thank the anonymous referees for their valuable comments on the original draft of this paper.

References

[BALLARIN 94] C. BALLARIN
Algorithmische Schritte in formalen Beweisen – Entwurf und Implementierung der Anbindung eines Computeralgebrasystems an einen Theorembeweiser. Diplomarbeit, Universität Karlsruhe, 1994.

[BÜNDGEN 94] R. BÜNDGEN
Combining Computer Algebra and Rule Based Reasoning. In [CALMET & CAMPBELL 94].

[CALMET & CAMPBELL 94] J. CALMET, J.A. CAMPBELL
Artificial Intelligence and Symbolic Mathematical Computing (AISMC-2), Editors, Lecture Notes in Computer Science, Springer-Verlag, 1994, to appear.

[CALMET & HOMANN 95] J. CALMET, K. HOMANN
Distributed Mathematical Problem Solving. In Proceedings of 4th Bar-Ilan Symposium on Foundations of Artificial Intelligence (BISFAI'95), 1995, to appear.

[CHAR et al. 92] B.W. CHAR, K.O. GEDDES, G.H. GONNET, B.L. LEONG, M.B. MONAGAN, S.M. WATT
Maple V Language Reference Manual, Springer-Verlag, 1992.

[CLARKE & ZHAO 94] E. CLARKE, X. ZHAO
Combining Symbolic Computation and Theorem Proving: Some Problems of Ramanujan. In A. BUNDY (Ed.), Automated Deduction (CADE-12), pp. 758–763, Lecture Notes in Artificial Intelligence 814, Springer-Verlag, 1994.

[MCCUNE 94] W.W. MCCUNE
OTTER 3.0 Reference Manual and Guide, Technical Report ANL-94/6, Argonne National Labaratory, 1994.

[GIUNCHIGLIA et al. 94] F. GIUNCHIGLIA, P. PECCHIARI, C. TALCOTT
Reasoning Theories – Towards an Architecture for Open Mechanized Reasoning Systems, 1994, to appear.

[HARRISON & THÉRY 93] J. HARRISON, L. THÉRY
Extending the HOL Theorem Prover with a Computer Algebra System to Reason About the Reals. In J.J. JOYCE, C.-J.H. SEGER (Eds.), Higher Order Logic Theorem Proving and its Applications (HUG'93), pp. 174–184, Lecture Notes in Computer Science 780, Springer-Verlag, 1993.

[HOMANN & CALMET 94] K. HOMANN, J. CALMET
Combining Theorem Proving and Symbolic Mathematical Computing. In [CALMET & CAMPBELL 94].

[JACKSON 94] P. JACKSON
Exploring Abstract Algebra in Constructive Type Theory. In A. BUNDY (Ed.), Automated Deduction (CADE-12), pp. 590–604, Lecture Notes in Artificial Intelligence 814, Springer-Verlag, 1994.

[KAJLER 92] N. KAJLER
CAS/PI: a Portable and Extensible Interface for Computer Algebra Systems. In Proceedings of ISSAC'92, pp. 376–386, ACM press, 1992.

[MILNER 85] R. MILNER
The use of machines to assist in rigorous proof. In C. HOARE, J. SHEPHERDSON (Eds.), Mathematical Logic and Programming Languages, pp. 77–88, Prentice-Hall, 1985.

[MILNER & TOFTE 91] R. MILNER, M. TOFTE
Commentary on Standard ML, MIT Press, 1991.

[PAULSON 83] L.C. PAULSON
A Higher-Order Implementation of Rewriting. In Science of Computer Programming 3, pp. 119–149, North Holland, 1983.

[PAULSON 94] L.C. PAULSON
Isabelle — A Generic Theorem Prover, Lecture Notes in Computer Science 828, Springer-Verlag, 1994.

[UEBERBERG 94] J. UEBERBERG
Interactive Theorem Proving and Computer Algebra. In [CALMET & CAMPBELL 94].

[WOLFRAM 91] S. WOLFRAM
Mathematica: a System for Doing Mathematics by Computer, Addison-Wesley, 1991.