Certified Computer Algebra on top of an Interactive Theorem Prover

Cezary Kaliszyk and Freek Wiedijk

{cek,freek}@cs.ru.nl
Institute for Computing and Information Sciences,
Radboud University Nijmegen, the Netherlands

Abstract. We present a prototype of a computer algebra system that is built on top of a proof assistant, HOL Light. This architecture guarantees that one can be certain that the system will make no mistakes. All expressions in the system will have precise semantics, and the proof assistant will check the correctness of all simplifications according to this semantics. The system actually *proves* each simplification performed by the computer algebra system.
Although our system is built on top of a proof assistant, we designed the user interface to be very close in spirit to the interface of systems like Maple and Mathematica. The system, therefore, allows the user to easily probe the underlying automation of the proof assistant for strengths and weaknesses with respect to the automation of mainstream computer algebra systems. The system that we present is a prototype, but can be straightforwardly scaled up to a practical computer algebra system.

1 Introduction

Computer algebra systems do not always give correct answers. This happens because those systems do not certify the operations performed. There can be various reasons for errors in a CAS: assumptions can be lost, types of expressions can be forgotten [2], the system might get confused between branches of 'multi-valued' functions, and of course the algorithms of the system themselves may contain implementation errors [23].

As an example of the kind of error that we are talking about here, consider the following Maple [11] session that tries to compute $\int_0^\infty \frac{e^{-(x-1)^2}}{\sqrt{x}} dx$ numerically in two different ways:

```
> int(exp(-(x-t)^2)/sqrt(x), x=0..infinity);
```

$$\frac{1}{2}\frac{e^{-t^2}\left(-\frac{3(t^2)^{\frac{1}{4}}\pi^{\frac{1}{2}}2^{\frac{1}{2}}e^{\frac{t^2}{2}}K_{\frac{3}{4}}(\frac{t^2}{2})}{t^2}+(t^2)^{\frac{1}{4}}\pi^{\frac{1}{2}}2^{\frac{1}{2}}e^{\frac{t^2}{2}}K_{\frac{7}{4}}(\frac{t^2}{2})\right)}{\pi^{\frac{1}{2}}}$$

```
> subs(t=1,%);
```

$$\frac{1}{2}\frac{e^{-1}\left(-3\pi^{\frac{1}{2}}2^{\frac{1}{2}}e^{\frac{1}{2}}K_{\frac{3}{4}}(\frac{1}{2})+\pi^{\frac{1}{2}}2^{\frac{1}{2}}e^{\frac{1}{2}}K_{\frac{7}{4}}(\frac{1}{2})\right)}{\pi^{\frac{1}{2}}}$$

```
> evalf(%);
```

$$0.4118623312$$

```
> evalf(int(exp(-(x-1)^2)/sqrt(x), x=0..infinity));
```

$$1.973732150$$

(We are showing MAPLE here, but all major computer algebra systems make errors like this.)

To be sure that results are correct, one may use a proof assistant instead of a CAS. But in that case even calculating simple things, like adding fractions or calculating a derivative of a polynomial becomes a non-trivial activity, which requires significant experience with the system.

Our approach is to implement a computer algebra system on top of a proof assistant. For our prototype we chose the LCF-style theorem prover HOL LIGHT [16]. Thanks to this, we obtain a CAS system where the user can be sure of the correctness of the results. Such a system has strong semantics, that is all variables have types, all functions have precise definitions in the logic of the prover and for every simplification there is a theorem that ensures the correctness of this simplification.[1] The interface of our computer algebra system resembles most CAS systems. It has a simple read-evaluate-print loop. The language of the formulas typed into the system is as close as possible to the language in which formulas are generally entered in CAS and to the language in which mathematics is done on paper. Interaction with the system currently looks like this[2]:

```
 In1 := (3 + 4 DIV 2) EXP 3 * 5 MOD 3
Out1 := 250
 In2 := vector [&2; &2] - vector [&1; &0] + vec 1
Out2 := vector [&2; &3]
 In3 := diff (diff (λx. &3 * sin (&2 * x) + &7 + exp (exp x)))
Out3 := λx. exp x pow 2 * exp (exp x) + exp x * exp (exp x) +
        -- &12 * sin (&2 * x)
 In4 := N (exp (&1)) 10
Out4 := #2.7182818284 + ... (exp (&1)) 10 F
 In5 := 3 divides 6 ∧ EVEN 12
Out5 := T
 In6 := Re ((Cx (&3) + Cx (&2) * ii) / (Cx (-- &2) + Cx (&7) * ii))
Out6 := &8 / &53
```

[1] In HOL LIGHT *simplification* is implemented through what in the LCF world is called *conversions*. A conversion is a function that takes a term and returns an equational theorem. The theorem has the given term on its left side and a simplified version of the term on the right side.

In this paper 'simplification' should *not* be taken to be a fixed reduction hard-wired into the logic of the proof assistant, the way it is in type theoretical systems like COQ [12].

[2] The '`&`', '`Cx`' and '`#`' are coercions to real, complex and floating point numbers

```
 In7 := x + &1 - x / &1 + &7 * (y + x) pow 2
Out7 := &7 * x pow 2 + &14 * x * y + &7 * y pow 2 + &1
 In8 := sum (0,5) (λx. &x * &x)
Out8 := &30
```

One can distinguish three categories of systems that try to fill the gap between computer algebra and proof assistants:

- Theorem provers inside computer algebra systems:
  - Analytica [6],
  - Theorema [8],
  - RedLog [13],
  - logical extension of Axiom [20].
- Frameworks for mathematical information exchange between systems:
  - MathML [10],
  - OpenMath [15],
  - OMSCS [7],
  - MathScheme [9],
  - Logic Broker [1].
- Bridges between theorem provers and computer algebra systems, also referred to as ad-hoc information exchange solutions:
  - PVS and Maple [14],
  - HOL and Maple [17],
  - Isabelle and Maple [4],
  - NuPrl and Weyl [18],
  - Omega and Maple/GAP [21],
  - Isabelle and Summit [3].

An important distinction that one can make within the category of bridges is the *degree of trust* between the prover and the CAS. In some of these solutions the prover uses the simplification of the CAS as an axiom, i.e., without checking its correctness. But in other solutions the prover takes the CAS output and then builds a verified theorem out of it. In this case there are again two possibilities: either the result is verified independently of how the CAS obtained it, or the system takes a trace of the rules that the CAS applied, and then uses that as a suggestion for what theorems should be used to construct a proof of the result.

In the work that we referred to here either the proof assistant is built inside the CAS, or the proof assistant and the CAS are next to each other. In our work however, we have the CAS inside the proof assistant.

Of course in many proof assistants there already is CAS-like functionality, in particular many proof assistants have arithmetic procedures or even powerful decision procedures. However, we do not just provide the functionality, but also build a *system* that can be used in a similar way as most other computer algebra systems are used.

In our system it is the first time that anyone pursued the combination of a CAS *inside* a proof assistant (in which all simplifications are validated), with an interface that has the customary CAS look and feel.

Our way of combining theorem proving and computer algebra has advantages over the ones presented above. All calculations done by our system are certified by the architecture of our system. All formulas defined inside it have types assigned, all defined operators have explicit semantics and all simplifications performed have theorems associated with them. No translation of formulas or semantics is needed, as the CAS shares the internal data structures of the proof assistant. There is no need to worry about mistakes in the implementation of the CAS, since all conversions are certified using the logic of the underlying prover. There is no verification required after the result is obtained, thanks to the creation of theorems alongside with the results. All simplifications performed by our architecture are completely certified, that is if a certificate for a particular simplification does not exist [5] it can not be performed. All variables used in HOL Light conversions have to be typed, so working in a proof assistant might seem less flexible than a traditional CAS implementation, but the abundance of decision procedures for HOL show that this probably is not a strong limitation.

The paper is organized as follows: in Section 2 we present the architecture of the system. In Section 3 we talk about the knowledge base. Finally in Section 4 we present a conclusion.

2 Architecture

We present a general architecture for a certified computer algebra system, and we will describe an implementation prototype. The source for the prototype is available from `http://www.cs.ru.nl/~cek/holcas/`. For the implementation we chose the proof assistant HOL Light [16]. The factors that influenced our choice were: the possibility to manipulate terms to create the conversions, prove theorems and implement the system in the same language[3], as well as a good library of analysis and algebra. The system created is rather a proof of concept than a real product, which is why the efficiency of the underlying prover was not a decisive factor. In particular we perform all computations inside the proof assistant's logic, sometimes with the help of decision procedures.

Our system is divided in three independent parts (Fig. 1): the user interface (input-response loop), the abstract algorithm of dealing with a formula (we will call this *the CAS conversion*), and the knowledge that is specific to the CAS system. That architecture allows the user both to use it as a computer algebra system, as well as making it usable in the context of theorem proving[4].

2.1 Input-response loop

The system displays a prompt, where one can write expressions to be simplified and commands. It is necessary to distinguish expressions to be computed or simplified from commands that represent actions that do not evaluate anything, like listing theorems or modifying and printing assumptions.

[3] HOL Light is written in OCaml and is provided as an extension of it

[4] *The CAS conversion* can be applied to a goal to be proved using `CONV_TAC`.

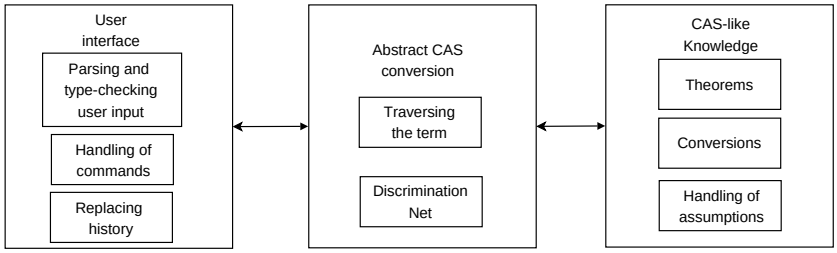

Fig. 1. Architecture of a CAS inside a TP system with responsibilities of the parts of our implementation marked. The prover is not marked on the figure, since all parts make use of it, by using it's type of terms and theorems, as well as tactics and conversions to build them.

Every expression that is not recognized as a command is passed to *the CAS conversion*, which will try to compute or simplify the expression. The theorem given back by *the CAS conversion* is the certificate that the output is correct. If *the CAS conversion* is not able to simplify the term, it returns an instance of reflexivity, and the result is then the same as the input.

In most CASs variables can be used without declaring them, but for certain algebraic operations one can define a variable to be of a particular type (necessary for example in Magma). Our system can handle expressions in both ways. The free variables are typed using HOL Light type inference, but one can also require a specific type with the `assumetype` command (described in section 3.4).

Most computer algebra systems allow one to reuse previously typed in expressions and calculated outputs. One may calculate `In1 + Out2`. The loop has to have access to all expressions entered, theorems proved and outputs. In our framework every expression entered is stored with its type, so when it is reused, parsing the same expression, even in a different context, gives the same type.

2.2 Abstract CAS conversion

To be able to benefit from the CAS simplification in theorem proving, it is useful to have the CAS functionality available as a single conversion (that we call here *the CAS conversion*). Since the underlying prover can be further developed and new theorems can be proved later, it is useful to separate *the CAS conversion* from the knowledge that it uses. For this reason *the CAS conversion* is parametrized. The general idea behind *the CAS conversion* is to try to apply all sub-conversions from the knowledge base at all positions in the term until it is saturated (Fig. 2). Applying the same conversions to a modified term is necessary, since some conversions return terms, parts of which can be again simplified. Particular implementations may include mechanisms to increase efficiency or to provide termination of simplification.

We are not particularly aiming at completeness for the algorithms in *the CAS conversion*, since completeness in practice can only be realized for rather basic theories. However any mathematically correct algorithm that exists for existing

computer algebra systems can be implemented as a HOL Light conversion too, that does the calculation while building the correctness proof in parallel. Examples include conversions that perform algorithms for integration, conversions that perform splitting and joining for branching calculations, or conversions that simplify terms involving higher order operations (like summation).

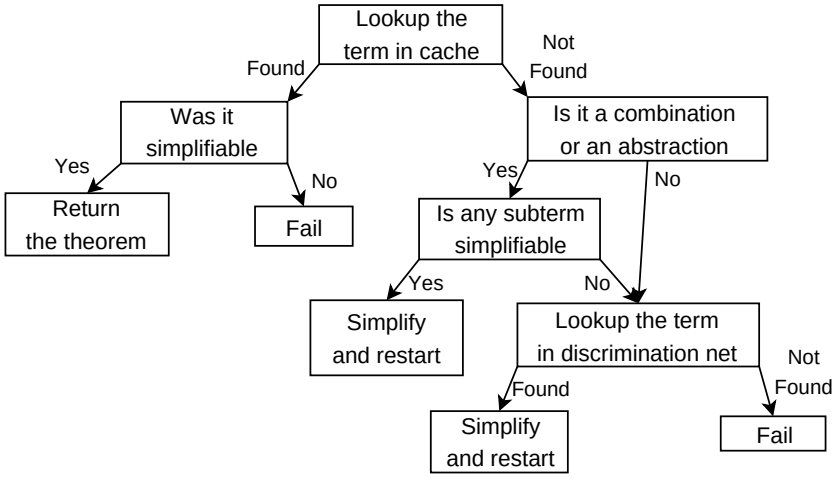

Fig. 2. Our implementation of *the CAS conversion* first tries to look up the term in a built-in cache (for efficiency). If the term is an application or an abstraction, then it tries to simplify subterms recursively (not performed if the term is known not to be expandable or is a suggestion that should not be expanded, for example `NUMERAL` or `assuming`). Finally it tries to apply all conversions from the knowledge base to the term.

3 CAS-like knowledge

The knowledge base is a separate part of the system. The knowledge is kept in a discrimination net (a structure that allows matching a term to a number of patterns efficiently). There is an interface on the theorem prover level that allows introducing knowledge to the knowledge base in the following three forms:

- Rewrite rules, for example:
  `|- ∀z. abs (norm z) = norm z`
- Conditional rewrite rules, for example:
  `|- ∀x. &0 <= x ==> abs x = x`
- Conversions meant to be used with an argument that matches a certain pattern and return ad-hoc rewrite rules. An ad-hoc rewrite rule is a theorem that is generated to be used for rewriting the formula, but it is not added

to the knowledge base (although our implementation keeps all rewritten theorems in cache, implemented as a hash-table, for efficiency reasons). For example the HOL Light conversion `DIVIDES_CONV` takes terms that match the pattern `n divides m` and then returns ad-hoc rewrite rules for the given data like `|- 33 divides 123453 <=> T`.

The CAS conversion has to check whether the given term matches one of the rewrite rules and ad-hoc rewrite rules in the knowledge base. For efficiency it keeps all theorems and conversions included in the knowledge base in a discrimination net. To allow matching conversions with even less overhead, optional patterns for matching associated with conversions can be provided. The discrimination net is not changed, the particular used instances are only added to the cache.

To resemble a CAS system, the formulas processed by the system should be in the "evaluation" form and not in "verification" form.

Let us compare the ways in which one writes differentiation in the HOL Light library and the way it is written in our CAS:

```
∀x. (f diffl (g x)) x      →     diff f = g
(f diffl (g x)) x          →     diff f x = g x
```

In HOL Light the `diffl` predicate takes three arguments: the function (on the left of the predicate), the value of its derivative and the point. To write a general derivative we need to generalize the point and replace the value with the derivative function in this point. Even then it is still a binary predicate.

In most computer algebra systems there exists a simple `diff` operator, that takes a function and returns its derivative. Using the Hilbert's choice operator, we created a such function, defined: `diff f = λx. εv. (f diffl v) x`. We also created a conversion that is able to calculate the derivative of a function, if HOL Light's `DIFF_CONV` can.

Just like we defined a functional form of differentiation, we also defined a functional integration operator. Using these we can then compute the following expressions in the system:

```
 In9 := dint (&1,&2) sin
Out9 := -- &1 * cos (&2) + cos (&1)
 In10 := dint (&1,&2) (λx. x pow n)
Out10 := &1 / &(n + 1) * &2 pow (n + 1) +
         -- &1 * &1 / &(n + 1) * &1 pow (n + 1)
 In11 := diff (diff (λx. &3 * sin (&2 * x) + &7 + exp (exp x))) (&2)
Out11 := exp (exp (&2)) * exp (&2) pow 2 + exp (exp (&2)) * exp (&2) +
         -- &12 * sin (&4)
 In12 := diff (λx. dint (x,x + &2) (λx. x pow 3))
Out12 := λx. &6 * x pow 2 + &12 * x + &8
```

Our differentiation and integration definitions do not work well with partial functions. An approach for defining them in such a way that partial functions are handled better will be described in Section 4.

3.1 Numerical approximations

In complex calculations computer algebra systems provide users with numerical approximations. They are usually implemented with an approximation algorithm, which keeps an error bound with every calculation. In a proof assistant a numerical approximation must have its semantics completely defined, and the algorithm has to respect the approximation definition and theorems that specify its properties.

The two main ways of rounding a real number are down to an integer and towards the nearest integer. Both these operations do not give rise to a computable function (see for example [19]). In [22] it is shown that if one computes non-deterministically either one of those values then one does get a computable function. We will use a conversion that calculates the value rounded both down and to nearest value, that terminates when one of those calculations terminate.

We propose to define the numerical approximation of a given number `x` to a precision `p` as identical to the number itself: `N x p = x`. It is only a hint for the system that the number has to be simplified to a decimal fraction plus a rest. It is the rest, that determines in which form is the number given: rounded down or rounded to the nearest. For rest defined in this way we provide a theorem, that states that the approximation can be different from the exact value only on the last digit, and the difference is less than one.

In the following HOL Light definitions, `N` is the numerical approximation of a number to a precision (following the convention of Mathematica) and `...` is the rest of a number to the given precision with an additional argument that specifies the form of the rest. `T` stands for rounding to nearest and `F` stands for rounding down.

```
... x p F = x - floor (&10 pow p * x) / &10 pow p
... x p T = x - floor (&10 pow p * x + &1 / &2) / &10 pow p
|- abs(... x p v - x) < &1 / &10 pow p
```

The system is able to compute some numerical approximations with this scheme:

```
 In13 := N (&1 / &3) 8
Out13 := #0.33333333 + ... (&1 / &3) 8 F
 In14 := N (sqrt #5.123456789) 8
Out14 := #2.26350542 + ... (sqrt #5.123456789) 8 F
 In15 := N (dint (#0.1,#0.4) exp) 7
Out15 := #0.3866537 + ... (-- &1 * exp #0.1 + exp (&2 / &5)) 7 F
```

3.2 Assumptions

In most CASs there is a possibility to make type assumptions or binary assumptions about variables. Examples include assuming a variable to be greater than zero, greater than another variable, natural or real. There are various methods of introducing assumptions in computer algebra systems:

- Assumptions associated with a simplification
  in MATHEMATICA: `Simplify[Sin[n Pi], Element[n,Integers]]`
- Global list of assumptions
  in MAPLE: `assume(x>0); sqrt(x*x);`
- Asking the user for conditions on variables (e.g. MAXIMA)
- Adding assumptions automatically and silently to the prover environment (e.g. MATHXPERT)

In our system we keep a global list of assumptions, which are Boolean properties that may be later used to instantiate assumptions of rewrite rules and ad-hoc rewrite rules. In a big CAS the number of rules that can be used is so big that asking the user seems not to be a good choice. Also automated assuming will probably not behave too well in such a situation.

An assumption can be introduced by the user either using `assume`, which takes a Boolean, or `assumetype` which takes a typed variable. An assumption associated with a single simplification of a sub-term may be also introduced using `assuming`. The latter method temporarily changes the assumptions list to simplifying the sub-expression. The assumptions will be added to the assumptions of the theorem generated by *the CAS conversion*, which is why changing the assumptions list is only useful at the top-level of the expression to simplify.

The global list of assumptions is used by the conversions from the knowledge base, therefore we consider is a part of the latter. To ensure the usage of variables with correct types, type checking has to have access to this list. When an expression is typed in the system it is type-checked in a particular context. This context includes types already assigned to all free variables from the assumptions list, as well as all variables for which types have been assumed with `assumetype`. To do this, the latter are kept in another global list.

For example, $\sqrt{x^2}$ cannot be simplified to x, since we don't know whether x is positive or not. Also $\frac{x}{x}$ cannot be simplified to 1, since it is possible that $x = 0$.

```
 In16 := sqrt (x * x)
Out16 := abs x
 In17 := x / x
Out17 := x * inv x
```

When an assumption about x is introduced, stating that it is greater then 1, numeric things about x can be proved, and both of the above formulas can be simplified:

```
 In18 := assume (x > &1)
Out18 := T
 In19 := x > #0.5
Out19 := T
 In20 := sqrt (x * x)
Out20 := x
 In21 := x / x
Out21 := &1
```

There are two ways in which assumptions are used: direct and indirect. The first way is to use an assumption directly in the derivation in unchanged form. It can be used to a prove a reflexive theorem or to fill the requirement of a certain conditional rewrite rule (or a conditional ad-hoc rewrite rule). An assumption may be used as an indirect step in the derivation, for example simplifying $abs(x)$ to x requires $x \geq 0$, and the assumption $x > 1$ can be used for this.

3.3 Manipulating assumptions

A CAS has to provide a mechanism for adding assumptions and listing defined assumptions. In our prototype we added the `assumptions` and `about` commands, which resemble their MAPLE equivalents.

```
Command: about Argument: x
  ‘x > &1‘
```

An issue that is hard to handle in any approach are errors that may be caused by incorrect parsing and printing. We try to be as close as possible to the original HOL LIGHT's parsing and printing mechanism. In fact, the system currently uses HOL's term printing (with special output for errors) but, when parsing, the system has to add typing information and distinguish commands from terms. Special output is added, so that the user always knows when a given string has been interpreted as a command.

To further lower the risk of parsing and printing problems, we add the `theorems` command. It allows printing all theorems defined in a session. The standard HOL LIGHT theorem printing function is used for this. It is especially useful for conversions that use assumptions, because in that it case the assumptions that have been actually used will be shown. Below are the first five theorems proved by the examples from this document:

```
Command: theorems
  |- (3 + 4 DIV 2) EXP 3 * 5 MOD 3 = 250
  |- vector [&2; &2] - vector [&1; &0] + vec 1 = vector [&2; &3]
  |- diff (diff (λx. &3 * sin (&2 * x) + &7 + exp (exp x))) =
       (λx. exp x pow 2 * exp (exp x) + exp x * exp (exp x) +
       -- &12 * sin (&2 * x))
  |- N (exp (&1)) 10 = #2.7182818284 + ... (exp (&1)) 10 F
  |- 3 divides 6 ∧ EVEN 12 <=> T
```

4 Conclusion

Our work integrates computer algebra and proof assistant technology. We will now look at how our architecture compares with what one gets by just having a CAS or a proof assistant.

Developing a system according to our architecture (i.e., where the algorithms not only generate the results, but also generate certificates of the correctness of

those results) will be slower than the development of traditional CAS systems (because that only has to generate the results). As far as the performance of the system is concerned, our architecture will be somewhat slower than a traditional CAS as well. This is mostly because generating the certificates for all simplifications also takes time. However, we expect this slow-down over traditional CAS to only multiply the running time by a constant factor. Our expectation is not experiment based, but based on the architecture, we trace what a traditional CAS does and provide proofs for every step.

When we compare our architecture to the way that one normally does CAS-like manipulations in an interactive theorem prover, the main difference is the interaction model. Our CAS system does not interactively work on propositions that are to be proved, but instead takes an expression and automatically simplifies it.

Our primary focus is to extend the knowledge base with a formalization of multivalued functions, to be able to handle more complicated expressions, like the MAPLE example of a complex function with multiple branches given in the introduction.

Another important feature that we plan to investigate are the coercions that many proof assistants use, like the embedding of the integers in the real numbers or the real numbers in the complex numbers. Currently a user of our prototype needs to use the '`&`' and '`Cx`' symbols for this (as is customary in the HOL LIGHT library). A small improvement to the current situation might be to overload the '`&`' operator, but we would rather not make the user write these functions at all.

An issue that our approach does not cover is completeness of the conversions. In the case of rewrite rules the completeness is clear. But in the case of arbitrary algorithms, it is not guaranteed by our architecture that a given conversion will always terminate and never fail.

We believe that both computer algebra systems and proof assistants currently have a problem. In computer algebra the lack of explicit semantics and the lack of verification of the results inside the system makes the systems less reliable than one would like them to be. In proof assistants the powerful symbolic manipulations that are taken for granted in computer algebra often are missing and, even when present, it takes work and expertise to make use of it.

We claim that the architecture that we present here can solve both problems simultaneously. The computer algebra systems will get explicit semantics and certification. And the proof assistants will get CAS-like functionality that will make them more powerful and easier to use than they are today.

References

1. A. Armando and D. Zini. Towards interoperable mechanized reasoning systems: the logic broker architecture. In A. Corradi, A. Omicini, and A. Poggi, editors, *WOA*, pages 70–75. Pitagora Editrice Bologna, 2000.
2. H. Aslaksen. Multiple-valued complex functions and computer algebra. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 30(2):12–20, June 1996.

3. C. Ballarin and L. C. Paulson. A pragmatic approach to extending provers by computer algebra - with applications to coding theory. *Fundam. Inf.*, 39(1-2):1–20, 1999.
4. Clemens Ballarin, Karsten Homann, and Jacques Calmet. Theorems and algorithms: an interface between Isabelle and Maple. In *ISSAC '95: Proceedings of the 1995 international symposium on Symbolic and algebraic computation*, pages 150–157, New York, NY, USA, 1995. ACM Press.
5. H. Barendregt and A. M. Cohen. Electronic communication of mathematics and the interaction of computer algebra systems and proof assistants. *J. Symb. Comput.*, 32(1/2):3–22, 2001.
6. A. Bauer, E. M. Clarke, and X. Zhao. Analytica - an experiment in combining theorem proving and symbolic computation. *Journal of Automated Reasoning*, 21(3):295–325, 1998.
7. P. Bertoli, J. Calmet, F. Giunchiglia, and K. Homann. Specification and integration of theorem provers and computer algebra systems. *Fundam. Inform.*, 39(1-2):39–57, 1999.
8. B. et al Buchberger. The Theorema Project: A Progress Report. In M. Kerber and M. Kohlhase, editors, *Symbolic Computation and Automated Reasoning (Proceedings of CALCULEMUS 2000)*, Natick, Massachusetts, 2000. A.K. Peters.
9. J. Carette, W. Farmer, and J. Wajs. Trustable communication between mathematics systems. In *CALCULEMUS 2003*, pages 55–68, Rome, Italy, 2003. Aracne.
10. David Carlisle, Patrick Ion, Robert Miner, and Nico Poppelier. Mathematical Markup Language (MathML) Version 2.0 (Second Edition), 2003.
11. B.W. Char, K.O. Geddes, W.M. Gentleman, and G.H. Gonnet. *The design of Maple: A compact, portable and powerful computer algebra system*. Springer-Verlag London, UK, 1983.
12. Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.0*. INRIA-Rocquencourt, January 2005.
13. A. Dolzmann and T. Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, June 1997.
14. A. Adams et al. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In R. J. Boulton and P. B. Jackson, editors, *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2001)*, volume 2152 of *Lecture Notes in Computer Science*, pages 27–42, Edinburgh, Scotland, UK, September 2001. Springer-Verlag.
15. S. Buswell et al. The OpenMath Standard, version 2.0, 2002.
16. J. Harrison. HOL light: A tutorial introduction. In M. Srivas and A. Camilleri, editors, *Proceedings of FMCAD'96*, volume 1166 of *LNCS*, pages 265–269. Springer-Verlag, 1996.
17. John Harrison and Laurent Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.
18. P. B. Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra*. PhD thesis, Cornell University, Ithaca, NY, USA, January 1995.
19. D.R. Lester. Effective continued fractions. In *Proceedings 15th IEEE Symposium on Computer Arithmetic*, pages 163–170. IEEE Computer Society Press, June 2001.
20. E. Poll and S. Thompson. Adding the axioms to Axiom: Towards a system of automated reasoning in Aldor. In *Calculemus and Types '98*, July 1998.
21. V. Sorge. Non-trivial symbolic computations in proof planning. In *FroCoS '00: Proceedings of the Third International Workshop on Frontiers of Combining Systems*, pages 121–135, London, UK, 2000. Springer-Verlag.

22. J. Vuillemin. Exact real computer arithmetic with continued fractions. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 14–27, New York, NY, USA, 1988. ACM Press.
23. M. J. Wester, editor. *Contents of Computer Algebra Systems: A Practical Guide*, chapter A Critique of the Mathematical Abilities of CA Systems. John Wiley & Sons, Chichester, United Kingdom, 1999.