In Memoriam: Chionh Eng Wee

Dr. Chionh Eng Wee, a Computer Algebraist and Associate Professor at National University of Singapore (NUS), died in September 2009.

Eng Wee received his B.Sc. (First Class Honours, 1974) from Nanyang University, M.Sc (1978) and Ph.D (1990) from the University of Waterloo. Before he joined NUS in 1984, he was a Systems Engineer at IBM (Singapore). His research interests included Computer Algebra and computer curves and surfaces.

He was involved with the National Olympiad in Informatics (NOI) in Singapore and in fact he was the Chair of the NOI 2006 Committee. See all http://www.comp.nus.edu.sg/~noi/

He was a participant at the recent MICA 2008 conference http://www.orcca.on.ca/conferences/mica2008 where he contributed his work titled “The Maximality of Dixon Matrices on Corner-Cut Monomial Supports by Almost-Diagonality”.

A personal tribute to Dr. Chionh Eng Wee is contained in the blog

We reproduce here a list of his publications taken from his webpage http://www.comp.nus.edu.sg/~chionhew/

2009

2008
2006

2005

2004

2003

2002

2001

2000

1999

1998

1997
On the Macaulay Inverse System and its Importance for the Theory of Linear Differential Equations with Constant Coefficients

By Wolfgang Gröbner in Rome

When first studying Macaulay’s inverse systems (for this, see [2, p. 65ff] first, then [3, page 37ff], and the excellent short exposition [1, page 66ff]), one can easily fall for the mistake that this is a difficult to understand, artificial construction of little value. Subsequent applications, which lead to a very surprising new insight, teach us better. In what follows, it will be shown how this shortcoming, insofar as it even exists, can be corrected in a simple way that draws on a new interpretation of the polynomial ring, which may be useful in and of itself. Macaulay’s inverse system is thereby connected to the system of integrals of linear homogeneous differential equations with constant coefficients and loses its character of possible foreignness. At the same time, this work represents an attempt to establish relationships between two seemingly completely separate branches of mathematics, which may be useful and fruitful for both sides.

1 The Inverse System

Let \( P_n = K[x_1, \ldots, x_n] \) be a polynomial ring over the base field \( K \), which we will assume from now on is the field of complex numbers, and let \( I = \langle f_1(x), \ldots, f_r(x) \rangle \) be an ideal in \( P_n \).

Let \( u = \sum c_{p_1,\ldots,p_n} x_1^{-p_1} \cdots x_n^{-p_n} \) be a formal power series which runs through negative powers of the variables \( x_1, \ldots, x_n \) and has coefficients \( c_{p_1,\ldots,p_n} \) in \( K \). If \( f(x) \) is any polynomial in \( P_n \), then the product \( f \cdot u \) defines the power series that results from formal multiplication, where we omit all those terms in which the exponent of one of the variables is positive. Then the inverse system \( I^{-1} \) belonging to the ideal \( I \) consists of all power series \( u \) of the type indicated, for which \( f \cdot u = 0 \) is omitted whenever \( f \in I \). For every proper ideal \( I \subset P_n \), power series exist which satisfy this condition. Then in particular, the following statements hold:

1. \( I^{-1} \) is a module with operator domain \( P_n \), i.e. for every finite sum (as usual, we omit the summation symbols), if \( g_i \in P_n \) and \( u_i \in I^{-1} \), then \( g_i u_i \in I^{-1} \). (This follows easily from \( f \cdot g_i u_i = g_i \cdot f u_i = 0 \) for \( f \in I \).)

2. If \( I \) and \( J \) are ideals in \( P_n \) and \( I^{-1} \) and \( J^{-1} \) are their respective inverse systems, then the following symbolic equations hold:

\[
I + J = I^{-1} \cap J^{-1}, \quad I \cap J = I^{-1} + J^{-1}, \quad I : J = J \cdot I^{-1}
\]

(i.e. \( I^{-1} \cap J^{-1} \) is the inverse system of the ideal \( I + J \), etc. [1].)

2 The Ring of All Linear Homogeneous Differential Equations with Coefficients in \( K \)

To every polynomial \( p(x) \in P_n \), we assign a unique differential operator \( p \left( \frac{\partial}{\partial x} \right) \), in which we just replace individual power products \( x_1^{i_1} \cdots x_n^{i_n} \) in \( p(x) \) by the symbols \( \frac{\partial^r}{\partial x_1^{i_1} \cdots \partial x_n^{i_n}} \), \( i = i_1 + i_2 + \ldots + i_n \), denoted \( p(x) \leftrightarrow p \left( \frac{\partial}{\partial x} \right) \) for short.

If in addition \( q(x) \leftrightarrow q \left( \frac{\partial}{\partial x} \right) \), then it follows easily that

\[
p(x) + q(x) \leftrightarrow p \left( \frac{\partial}{\partial x} \right) + q \left( \frac{\partial}{\partial x} \right)
\]

\[
p(x) \cdot q(x) \leftrightarrow p \left( \frac{\partial}{\partial x} \right) \cdot q \left( \frac{\partial}{\partial x} \right).
\]

Since the base field \( K \) remains fixed element-wise under this mapping, the two domains \( P_n = K[x_1, \ldots, x_n] \) and \( D_n = K \left[ \frac{\partial x_1}{\partial x}, \ldots, \frac{\partial x_n}{\partial x} \right] \) differ only in the different notation of their transcendental elements, and are therefore isomorphic. Thus we can immediately transfer all of the terms and theorems of ideal theory from the polynomial ring \( P_n \) to the differential ring \( D_n \). However, we must keep in mind that the indeterminates \( x_1, \ldots, x_n \) in \( P_n \) may not be interchanged with the variables \( x_1, \ldots, x_n \) in \( D_n \); only the formal mapping \( x_i \leftrightarrow \frac{\partial}{\partial x_i} \) stands between them.

3 The Inverse System or Integral System of an Ideal \( I \) in the Differential Ring \( D_n \)

In particular, we could carry over the definition of inverse systems to \( D_n \), as in Section 1, where we would want to establish, somewhat axiomatically, differential symbols that would correspond to negative powers of \( x \). However, another alternative is available here, which is even compatible with inverse systems in the true sense. If \( u = \sum c_{p_1, \ldots, p_n} x_1^{-p_1} \cdots x_n^{-p_n} \) is a formal power series as in Section 1, then we first want to rewrite it in the form

\[
\tilde{u} = \sum c_{p_1, \ldots, p_n} x_1^{p_1} \cdots x_n^{p_n} / p_1! \cdots p_n!.
\]

(Macaulay actually used this notation [2, page 73]), where we just replace \( x_i^{-p_i} \) with \( \frac{\partial}{\partial x_i} \). Then using the multiplication rule from Section 1, we simply declare that for arbitrary \( p(x) \in P_n \), \( p(x) \cdot u \) and \( p \left( \frac{\partial}{\partial x} \right) \) have the same result, except for different notation.

That the operation \( p \left( \frac{\partial}{\partial x} \right) \) fits with the conception of inverse systems shows however that we no longer need a specially tailored definition, but rather can get by with the elementary rules of differential computation. (As is well-known, all of the terms and rules of differential computation can be established and deduced purely algebraically; this means then that disciplines that are different in nature are not mixed with each other.) Thus we can define the following:

**Definition 1.** If \( I \) is a (proper) ideal of the differential ring \( D_n \), then the integral system \( I^{-1} \) denotes the set of all power series \( u = \sum c_{p_1, \ldots, p_n} x_1^{p_1} \cdots x_n^{p_n} \), \( (c_{p_1, \ldots, p_n} \in K) \), which are integrals of the ideal \( I \), i.e. for which \( p \left( \frac{\partial}{\partial x} \right) u = 0 \) holds whenever \( p \left( \frac{\partial}{\partial x} \right) \) lies in \( I \).

Then we have the following theorem:

**Theorem 3.1.** If the ideals \( I \) in \( P_n \) and \( \tilde{I} \) in \( D_n \) map to each other via the isomorphism \( x_i \leftrightarrow \frac{\partial}{\partial x_i} \), then the inverse system \( I^{-1} \) of \( I \) maps to the integral system \( \tilde{I}^{-1} \) of \( \tilde{I} \) by virtue of the mapping \( x_i^{-p_i} \leftrightarrow \frac{\partial}{\partial x_i} \).

Thus every theorem about the inverse system corresponds to a dual theorem about the integral system of an ideal of homogeneous linear differential equations, and conversely. In what follows, we offer some examples from the abundance of theorems that follow from this, without claiming any sort of completeness. This will show in fact that a vast area for algebraic research has opened up, which has apparently lain fallow until now.

4 Some Examples and Applications

First, statements 1 and 2 from Section 1 may be carried over to integral systems; we just wish to emphasize here that the integral system of an ideal \( I \) of \( D_n \) is a \( D_n \)-module, that therefore all derivatives of integrals are again integrals. The next theorem follows easily:
Theorem 4.1. A zero-dimensional ideal $I$ in $D_n$ has a finite number $r$ of linearly independent integrals, where $r$ is equal to the sum of the lengths of primary components of $I$. In particular, if every primary component of $I$ is irreducible, which is always the case when $I$ has a basis of precisely $n$ elements [2, page 81, §72], then $I$ has a principal integral $u$ such that every other integral can be represented linearly by $u$ and $r - 1$ suitably chosen derivatives of $u$.

Theorem 4.2. For a complete system (fundamental system) of linearly independent integrals $u_1, \ldots, u_r$ (of total length $r$) of a zero-dimensional ideal $I$ in $D_n$, the Addition Theorem holds:

$$u_i(x + y) = a_{kl}^i u_k(x) u_l(y), \quad (a_{kl}^i \in K)$$

(or alternately, $u_i(x_1 + y_1, \ldots, x_n + y_n) = \sum_{k,l=1}^r a_{kl}^i u_k(x_l)(y)$ (for $i = 1, 2, \ldots, r$)).

Proof. Clearly, $u_i(x + y)$ is also an integral of $I$, which we can formulate in complete generality:

Theorem 4.3. The integral system of an arbitrary ideal $I \subset D_n$ is invariant under translation of the space of variables.

Since $p \left(\frac{\partial}{\partial x_i}\right) u(x + y) = 0$ for all $p \left(\frac{\partial}{\partial x_i}\right) \in I$, it follows immediately that $u_i(x + y) = c_1^i(y)u_1(x) + \ldots + c_r^i(y)u_r(x)$. But $p \left(\frac{\partial}{\partial y}\right) u_i(x + y) = p \left(\frac{\partial c_1^i}{\partial y}\right) u_1(x) + \ldots + p \left(\frac{\partial c_r^i}{\partial y}\right) u_r(x) = 0$ for all $p \left(\frac{\partial}{\partial x_i}\right) \in I$, thus by the linear independence of the integrals $u_1, \ldots, u_r$, we have $p \left(\frac{\partial c_k^i}{\partial y}\right) = 0$, and hence $c_k^i(y) = a_{kl}^i u_l(y)$. 

A generalization of Theorem 3 is the following (this theorem already appeared in [2, page 73]):

Theorem 4.4. A homogeneous linear substitution of indeterminates $x_i$ corresponds to the transposed inverse substitution of variables $x_i$.

As is well-known, the symbols $\frac{\partial}{\partial x_i}$ substitute contragrediently for the variables $x_i$.

Theorem 4.5. If $e(x)$ denotes the normed ($e(0) = 1$) integral of the ideal $\langle \frac{\partial}{\partial x} - 1 \rangle$ in $D_1$ and $\{\xi_1, \ldots, \xi_n\}$ is any zero of an arbitrary ideal $I$ in $D_n$, then $e(\xi_1 x + \ldots + \xi_n x_n) = e(\xi_1 x_1) \cdots e(\xi_n x_n)$ is an integral of $I$.

Proof. $e(\xi_1 x_1 + \ldots + \xi_n x_n)$ is the integral of the zero-dimensional prime ideal $\langle \frac{\partial}{\partial x_1} - \xi_1, \ldots, \frac{\partial}{\partial x_n} - \xi_n \rangle = p$, and since $I \subset p$, an integral of $I$ as well.

Theorem 4.6. The numbers $a_{kl}^i$ in the base field $K$ defined in Theorem 4.2 determine a commutative algebra $A = K[e_1, \ldots, e_r]$, where multiplication of the elements $e_i$ is defined by $e_k e_l = a_{kl}^i e_i$.

Proof. We have $u_i(x + y) = a_{kl}^i u_k(x) u_l(y)$ and $u_i(y + x) = a_{kl}^i u_k(x) u_l(y)$. Since $u_i(x + y) = u_i(y + x)$ and by the linear independence of the integrals, $a_{kl}^i = a_{lk}^i$ follows, i. e. the multiplication table $(a_{kl}^i)$ is commutative. Furthermore,

$$u_i(x + y + z) = a_{kl}^i u_k(x) u_l(y + z) = a_{kl}^i a_{km}^l u_k(x) u_m(y) u_p(z) = a_{lp}^i u_l(x + y) u_p(z) = a_{lp}^i a_{km}^l u_k(x) u_m(y) u_p(z),$$

therefore $a_{kl}^i a_{lm}^p = a_{lp}^i a_{km}^l$, i. e. the associative law for multiplication is also satisfied.

Theorem 4.7. The following formulae hold (where $m = 1, 2, \ldots$):

$$[e_1 u_1(x) + \ldots + e_r u_r(x)] [e_1 u_1(y) + \ldots + e_r u_r(y)] = e_1 u_1(x + y) + \ldots + e_r u_r(x + y),$$

$$[e_1 u_1(x) + \ldots + e_r u_r(x)]^m = e_1 u_1(mx) + \ldots + e_r u_r(mx).$$

Proof. $[e_1 u_1(x)] [e_k u_k(y)] = e_k e_k u_1(x) u_k(y) = e_k a_{kl}^i u_k(x) u_l(y) = e_1 u_1(x + y).$
In the special case of the ideal \( \left\langle \frac{\partial^2}{\partial x^2} + 1 \right\rangle \) in \( D_1 \), we have the well-known formulae of De Moivre:

\[
(\cos x + i \sin x)(\cos y + i \sin y) = \cos(x + y) + i \sin(x + y),
\]

\[
(\cos x + i \sin x)^m = \cos mx + i \sin mx,
\]

which therefore represents a special case of a very general theorem. Perhaps this investigation would pay off if we could find a generalization of the formula

\[
(\cos x + i \sin x)(\cos x - i \sin x) = \cos^2 x + \sin^2 x = 1,
\]

and then we could perhaps discover, whether the integral of a zero-dimensional ideal produces parametric representations for certain algebraic manifolds, with further problems tied to it.

References


Abstracts of Invited Talks, Contributed Posters, and Software Demos at the East Coast Computer Algebra Day 2010

communicated by Vicky Powers

1 Invited Talks

Anton Leykin, Georgia Institute of Technology:

Certified numerical solving of systems of polynomial equations.

Polynomial homotopy continuation is at the heart of numerical algebraic geometry, an area whose primary goal is to solve systems of polynomial equations. Recently this field developed rapidly producing several software packages that solve some problems out of reach for purely symbolic techniques. However, the homotopy tracking procedures employ heuristics to follow the homotopy continuation paths. Jointly with Carlos Beltran we have devised a certified homotopy tracking algorithm, which was implemented in Macaulay2. In addition we conducted experiments shedding light on some developments in complexity analysis of polynomial systems solving.

Mark van Hoeij, Florida State University:

The complexity of factoring univariate polynomials over the rationals.

In 2001 a new algorithm was introduced for factoring polynomials over the rationals. The algorithm was soon adopted by several computer algebra systems because of its good practical performance. A complexity result for a slower version of the algorithm was given in a preprint [BHKS, 2004], but there remained a very large gap between the proven bound and the behavior of the implementations as observed in actual examples. This gap was closed in joint work with Andy Novocin. We give a modification of the algorithm that allows us to prove the best complexity bound for factoring in $\mathbb{Q}[x]$. Moreover, unlike [BHKS, 2004], an implementation by Novocin demonstrates that this modification does not slow down the algorithm in practice.

David Saunders, University of Delaware

Exact Linear Algebra

Most of the methods of linear algebra, even the iterative ones, are exact in principle but are applied approximately. Over the past half century numerical analysis has made great strides in the understanding of approximation in matrix computation. But also, roughly over the past three decades, a solid algorithmic basis for exact sparse and dense matrix computation has emerged. We argue that the computer algebra community has the interests, techniques, motivations, and applications to develop high performance computational capabilities for exact linear algebra.

It is a tale of two equivalence relations, similarity and matrix equivalence. These were characterized in the 19th century via the Jordan canonical form and Smith normal form respectively. Associated with these are the computational concerns of numerical linear algebra, eigenvalues and linear system solving respectively. But applications of exact linear algebra tend to need some or all of the invariants themselves associated with these equivalences. From rank to characteristic polynomial, computation of these invariants resists pure numerical solution and provides an important role for computer algebra.

We discuss the state of the art and current issues in algorithms and software system design for exact linear algebra over the integers, the rational numbers, and over finite fields. We show some of the applications in which large scale exact linear algebra has been used.
2 Contributed Posters

Gregory V. Bard, Fordham University:

DEMOCRACY. A Heuristic for Polynomial Systems of Equations over Finite Fields

This paper presents a heuristic for solving polynomial systems of equations on finite fields larger than GF(2), via stochastic local search (SLS). It was inspired by the SLS-based SAT-solvers G-SAT, Walk-SAT and Tabu-SAT. Called DEMOCRACY, the equations vote on which values for a given variable will satisfy as many as possible of them. Variables, one at a time, are thusly changed from an initial random setting, until all equations are satisfied.

Mark Giesbrecht, Daniel S. Roche, Hrushikesh Tilak, University of Waterloo:

Complexity of Sparsest Multiple Computation

We investigate the problem of computing the sparsest multiple of a given univariate polynomial. Over the rational numbers we exhibit algorithms for when the target sparsity is two (i.e., the existence of a binomial multiple), as well as some cases when the target multiple has greater (constant) sparsity. Over finite fields, we tie the cost of finding a binomial multiple (of low degree) to order-finding in the multiplicative group of a finite field, as well as presenting similar, but more limited results for general sparsity.

Deborah Mathews, University of Rhode Island:

An Empirical Study of Parallel Big Number Arithmetic

To capitalize on multi-core processing, it would be good to perform big number arithmetic in parallel. While a performance increase equal to the number of processors is theoretically possible, our experiments show that in practice the likelihood of gaining any performance increase for big number arithmetic through parallel processing is low. A speed-up approaching the number of processors was not achieved for multiplication until the operands had at least 2^{1.5} bits. No performance gain was realized for addition. The base algorithm used for multiplication was O(n^2) and, therefore, suboptimal. The expected performance gain achieved when parallelizing a more efficient base algorithm should be even smaller.

Lingchuan Meng, Jeremy Johnson, Franz Franchetti, Yevgen Voronenko, Marc Moreno Maza, Yuzhen Xie, Drexel University:

SPIRAL-Generated Modular FFTs

In this poster we present the use of the SPIRAL system (www.spiral.net) to generate code for modular Fast Fourier Transforms (FFTs). SPIRAL is a library generation system that automatically generates platform-tuned implementations of digital signal processing algorithms with an emphasis on fast transforms. Currently, SPIRAL can generate highly optimized fixed point and floating-point FFTs for a variety of platforms including vectorization, multi-threaded and distributed memory parallelization. The code produced is competitive with the best available code for these platforms and SPIRAL is used by Intel for its IPP (Intel Performance Primitives) and MKL (Math kernel Library) libraries.

The SPIRAL system uses a mathematical framework for representing and deriving algorithms. Algorithms are derived using rewrite rules and additional rules are used to symbolically manipulate algorithms into forms that take advantage of the underlying hardware. A search engine with a feedback loop is used to tune implementations to particular platforms. New transforms are added by introducing new symbols and their definition and new algorithms can be generated by adding new rules.

We extended SPIRAL to generate algorithms for FFT computation over finite fields. This addition required adding a new data type, several new rules and a new transform (ModDFT) definition. In addition, the unpars (where code is generated) was extended so that it can generate scalar and vectorized code for modular arithmetic. With these enhancements, the SPIRAL machinery can be applied to modular transforms that are of interest to the
computer algebra community. This provides a framework for systematically optimizing these transforms, utilizing vector and parallel computation, and for automatically tuning them to different platforms. In this poster we present preliminary results from this exploration. We show that the code generated by SPIRAL, with improved cache locality and vectorization, is approximately ten times faster than the modular FFT code in the modpn library.

Bryan Youse, B. David Saunders, University of Delaware:

**Bitslicing with Matrix Algorithms Oblivious to the Data Compression**

Bitslicing is a data compression technique useful for fast exact linear algebra over finite fields whose elements are significantly smaller than a single machine word. It involves striping the number of bits used to represent field elements over the same number of machine words. Thus, this data layout is maximally storage-efficient. We showcase the computational advantages. We also discuss practical challenges which arise when employing this scheme on large matrices and their sub-matrices over GF(3) with algorithms written to work generically with matrices over GF(q), for any q.

### 3 Software Demonstrations

Tingting Fang, Mark van Hoeij, Florida State University:

**Solving Linear Differential Equations by Using Descent**

For second order linear ordinary differential equations with coefficients in C(x), we present an algorithm to reduce, whenever possible, the equation to an equation defined over a subfield of C(x). At the moment, we have implemented 2-descent, which means that if there exists a reduction to a subfield of index 2, then we can find it. If n is the number of true singularities, then 2-descent, if it exists, reduces the number of true singularities to at most n/2 + 2. In 12 out of the 17 examples sent to us by Bostan and Kauers, we found that repeated use of 2-descent reduced their regular singular equation down to 3 singularities, which means that 2-descent allows us to find $2F_1$-hypergeometric type solutions for about two third’s of their equations.

Yongjae Cha, Mark van Hoeij, Giles Levy, Florida State University:

**Solving Linear Recurrence Relations**

We will present an implementation of several algorithms for solving second order linear recurrence relations. The algorithms are described in two papers accepted at ISSAC 2010. Our implementation can find Liouvillian solutions, as well as solutions written in terms of values of special functions such as the $2F_1$ hypergeometric function, Bessel, Whittaker, Legendre, Laguerre, etc.

We have done an automated search in Sloane’s online encyclopedia of integer sequences, to find sequences that satisfy a second order recurrence. Our implementation solves a large majority of such recurrence relations.

The papers and implementation are available at [http://www.math.fsu.edu/~glevy/implementation](http://www.math.fsu.edu/~glevy/implementation)
Emerging Trends Papers accepted for PLMMS 2010

Communicated by Lucas Dixon and James Davenport

The 4th International Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS-2010) was one of the Conferences in Intelligent Computer Mathematics (CICM) in Paris, France. It took place on the 8th of July.

The scope of the workshop is the intersection of programming languages and mechanized mathematics systems. This includes programming languages and aspects of present-day computer algebra systems, interactive proof assistants, and automated theorem provers, all heading towards fully integrated mechanized mathematical assistants.

We would like to thank the authors who submitted papers and the members of the Program Committee for their dedication and the constructive reviews.

Enjoy,

James Davenport, University of Bath
Lucas Dixon, University of Edinburgh
(PLMMS-2010 Programme Chairs)

CTP-based programming languages? — Considerations about an experimental design

Florian Haftmann\textsuperscript{a}  Cezary Kaliszyk\textsuperscript{a}  Walther Neuper\textsuperscript{b}

Abstract

This paper discusses plans for joint work in order to gain early feedback from the community.

Three lines of work pursued independently so far shall be joined: (1) narrowing the gap between declarative program specification and program generation already working in Isabelle, (2) reusing work, which embedded an input-response-loop resembling Computer Algebra Systems (CAS) into HOL Light, and (3) reconstructing an experimental language for applied mathematics by exploiting established as well as emerging features of Isabelle/Isar.

These plans have to be seen as part of a variety of highly active research areas — on “integration of the deduction and the computational power” of Computer Theorem Proving (CTP) and CAS respectively (Calculmus), on “innovative theoretical and technological solutions for content-based systems” (MKM), on “Programming Languages for Mechanized Mathematics Systems” (PLMMS), just to cite from some related interest groups.

Facing the abundant variety of approaches, of intermediate results and of ongoing developments, and taking under consideration the many difficulties in integrating such approaches, we pursue pragmatic goals:

Design a component indispensable for working engineers, a programming language for engineering applications. Use Isabelle for an experimental embedding of the language, which is useful at least in engineering education as soon as possible.

\textsuperscript{a}TU München - Software & Systems Engineering, \{haftmann,kaliszyk\}@informatik.tu-muenchen.de

\textsuperscript{b}TU Graz - Institute for Software Technology, neuper@ist.tugraz.at
1 Introduction

We take the term “computer theorem proving” as introduced in [13] comprising both, automated and interactive theorem proving. Both aspects of theorem proving are relevant for our approach. The abbreviation “CTP” for “computer theorem proving” shall indicate analogies to “CAS”, a widely used abbreviation for “computer algebra (systems)” 1.

Our motivation is simple: transfer the success from programming languages based on Computer Algebra Systems (called CAS-based languages) to the domain of Computer Theorem Proving (CTP), overcome deficiencies found basically and essentially in all CAS and improve safety and reliability of software by use of concepts and technologies from CTP. What is called a success of CAS-based languages is the fact 2 that a major and quickly increasing part of software for electrical engineering, for structural engineering and the like is built using such languages.

Our approach is pragmatic: we start from three lines of work pursued independently so far and discuss a merge of these. The three lines are: (1) narrowing the gap between declarative program specification and program generation already working in Isabelle (Sect.2.1), (2) reusing work on a CAS-like input-response-loop embedded into HOL Light (Sect.2.2), and (3) reconstructing an experimental language for applied mathematics (Sect.2.3) in the ISAC-prototype by exploiting the emerging features of Isabelle/Isar. And we confine our approach to the Isabelle framework.

Turning the idea for merging into a concrete research plan faces considerable issues: the idea is intimately interrelated with a variety of highly active research — with the “integration of the deduction and the computational power” of CTP and CAS respectively, with “innovative theoretical and technological solutions for content-based systems”, with “Programming Languages for Mechanized Mathematics Systems” (just to cite from some interest groups).

So, this paper does not give a comprehensive survey; only the most important concepts involved in the merge are addressed. And the paper goes into technical details only if it seems necessary for comprehension. Furthermore we do not feel ready to give a concise specification of the language envisaged. Rather, we pursue the above motivation and describe a future workplace of an engineer who constructs software of the kind presently covered by CAS-based languages in Sect.3. Continuing the pragmatic approach requires to mention practical aspects like the workflow at the future workplace, which necessarily remains speculative.

The paper is organized as follows: Sect.2 describes each of the three approaches and work already accomplished. Sect.3 tries an outlook to an engineers electronic workbench in the future in order to present, how the merge of the three approaches might come to bear. Sect.4 discusses novel research set on stage by the merge of the three approaches. Sect.5 relates the language under consideration with existing languages of proof assistants and mentions related work on the integration of reasoning and computation and on improving reliability of calculation with reals. Sect.6 gives a summary and an outlook to continuation of the presented work.

2 Three approaches and straightforward merges

The three lines of work have been pursued independently so far. They are presented with respect to possibilities for mutual merges together with advantages, which seem straightforward and which do not necessarily require remarkable R&D.

2.1 Integration of deductive and algorithmic components

Integration of deduction and calculation is being promoted from several sides (see Sect.5), from the side of programming languages, from symbolic computation and from the side of CTP. This paper takes the latter approach, based on Isabelle.

Isabelle [32] has been recognized as a logical framework [36] for a long time. With programming in mind one recognizes, that Isabelle provides generic numerals [35] and also floating point numbers [14]. Presently Isabelle/Isar’s logical infrastructure seems to develop towards a “logical operating system” [39] for various applications. The CTP-based language under consideration is one of such applications.

1Both terms, CTP and CAS, will be used to designate two different things: the respective software products and the respective underlying concepts and technologies.

2The most successful software house building engineering software on demand is Wolfram Research, see http://www.wolfram.com/solutions.
Bringing together specification and implementation. Recently the well-known relationship between higher-order logic and functional programming has been exploited [11]. The central idea is that a suitable set of equational theorems of the form $f \equiv t$ is interpreted as a functional program which can be translated to suitable languages like Haskell or ML. Given a function symbol $f$ with a specification $Pf$, the implementation of $f$ is constructed by deriving suitable equations $f \equiv t$ describing $f$ from $Pf$. Figuratively spoken, an implementation is a coagulation of equational theorems from the logic.

This approach allows to express refinement directly within the logic: e.g., imagine a function symbol $b\text{sort} :: \alpha \text{ list} \Rightarrow \alpha \text{ list}$ with corresponding equational theorems implementing bubble sort and a function symbol $m\text{sort} :: \alpha \text{ list} \Rightarrow \alpha \text{ list}$ with corresponding equational theorems implementing merge sort. Then $b\text{sort} = m\text{sort}$ can be proven, which allows to replace $b\text{sort}$ by $m\text{sort}$ in implementations. It would even be possible to specify sorting involving a choice operator, e.g., $\text{sort} \; \text{xs} = \text{THE} \; \text{ys}. \text{multiset} \; \text{xs} = \text{multiset} \; \text{ys} \land \text{sorted} \; \text{ys}$, where $\text{multiset}$ turns a list into the corresponding multiset. From this definition $\text{sort} = m\text{sort}$ (and $\text{sort} = b\text{sort}$) can be proven, hence the abstract $\text{sort}$ can be implemented by a concrete algorithm.

There are examples of engineering problems (for instance the one in Sect.2.3 on p.31) on which the above method seems to be applicable. In software development automated or semi-automated code generation is an appealing offer; in the application domain under consideration the major benefit might be in support for handling types in highly complex mathematical structures (rather than automated coding of complicated algorithms).

2.2 Approach towards CAS-like functionality

Before considering “CAS-like functionality” for CTP-based programming languages, we need to mention the deficiencies of mainstream CAS in order to be clear, what shall be improved when advancing to a CTP-based language.

Mainstream CAS, for instance Mathematica and Maple, are very weakly founded ([13] even calls them “ill-defined”). There are various reasons for the mistakes found in mainstream CAS systems: assumptions can be lost, types of expressions can be forgotten or algorithms of the system themselves may contain implementation errors [18]. Simple mistakes have been found and fixed over the years. However mistakes made when performing more complicated computations are still found. So improvements are urgently required, and our work already tackled some of them.

We have built a prototype CAS-like input-response-loop inside HOL Light, with the user interface designed close to the interfaces of popular computer algebra systems. In Figure 1 we show examples of simplifications that it can perform automatically: basic vector arithmetic, symbolic computation, numeric approximations and basic handling of assumptions.

```
In1 := vector [2; 2] - vector [1; 0] + vec 1
Out1 := vector [2; 3]

In2 := diff (diff (λx. 3 * sin (2 * x) + 7 + exp (exp x)))
Out2 := λx. exp x pow 2 * exp (exp x) + exp x * exp (exp x) + -- 12 * sin (2 * x)

In3 := N (exp (1)) 10
Out3 := #2.7182818284 + ... (exp (1)) 10 F

In4 := x + 1 - x / 1 + 7 * (y + x) pow 2
Out4 := 7 * x pow 2 + 14 * x * y + 7 * y pow 2 + 1

In5 := sum (0,5) (λx. x * x)
Out5 := 30

In6 := sqrt (x * x) assuming x > 1
Out6 := x
```

Figure 1: Example interaction with the prototype CAS-like input-response loop. For the user input given in the In lines, the system produces the output in Out lines together with HOL Light theorems that state the equality between the input and the output.

By this prototype we have demonstrated that it is possible to build a computer algebra system in a proof assistant [18]. Such architecture guarantees that the system will make no mistakes. All expressions in the system have precise semantics and the proof assistant checks the correctness of all simplifications according to this semantics.
The envisaged language shall be based on such a system; also CAS-like interaction as shown in Figure 1 shall be available for the user.

There are issues, which are still open in the prototype. First the syntax includes many coercions. In the presented example the symbol \& marks coercions to real numbers. HOL Light’s type prioritization is used to decide to which type the variables and operators (plus, ...) should belong; but the overloading there is not strong enough, although we are able to show quite some type information. In Sect.4.1 we describe how this can be solved using Isabelle’s proper parsing and syntax translation mechanisms.

Partiality: The prototype provides a simple mechanism for handling assumptions. In the example we have seen the construct assuming that allows the system to simplify $\sqrt{x^2}$, by deriving that $x$ is non-negative. And extension of this for handling partiality with functions is presented in [17]. This lets the prototype compute the derivative of $\frac{1}{x}$ knowing that $x \neq 0$.

The assumptions are stored in two lists. A list that stores assumptions about variable types and a list of properties. The first of those lists is given to the parser, while the second one is used to fulfill the assumptions of conditional rewrite rules. This means that the approach is built on top of the proof assistant. However, the integration is not perfect, since the decision procedures present in the proof assistant cannot make use of the assumptions. In Sect.4.1 we describe how Isabelle contexts can be used to combine the assumptions with the whole of the proof assistant.

The mechanism described above seems appropriate also for programs executing some application of mathematics. For this purpose this mechanism has to be integrated with mechanisms from Isabelle, see Sect.4.1. There are features indispensable in CAS like numerals discussed as well.

### 2.3 A functional language with guards

The third approach contributing to the envisaged joint work dates back to a language implementation [30] for educational purposes in the Z34c-prototype. Over the years this language revealed potential for generalization. The following features of the language are relevant for the envisaged joint work:

1. The language is based on Isabelle/HOL with IF, LET, IN and functions on lists like HD, TL, LAST etc. Straight forward extensions provide for access to Isabelle’s matching and rewriting. The language is purely functional (without input and output statements) and inherit major features from SML: strict evaluation, high-order functions, abstract datatypes, compile-time type checking and type inference; it might be statically nested with other functions. In comparison to SML it comes without a module system, without pattern matching for datatypes and without exception handling.

2. Functions (and also functions nested within other functions) are guarded with a formal specification, i.e with typed input- and output-items, precondition and postcondition. If the precondition, instantiated with the input-items’ values, holds, then the guard allows to start program execution. Patterns of specifications are given in a tree, and traversing the tree while matching the patterns with the input-items and evaluating the precondition allows to determine the most appropriate specification. This kind of “problem refinement” has successfully been used to model a simple equation solver [21] and to make other CAS-like functionality transparent.

3. The language comes along with an interpreter, which operates on the parse tree created by Isabelle’s parser. The initial purpose of the interpreter was to provide user guidance for the tutoring system: switched into a single-stepping mode the interpreter hands over control to a dialog module at certain steps and the dialog handles interaction with the learner.

With respect to the joint work envisaged here, the interpreter is relevant for another reason: it works on the constructs of the programming language on the same level of abstraction, as Isabelle’s prover works on constructs of the specification, i.e on terms and predicates.

---

3The essential design ideas were provided by Peter Lucas, one of the pioneers in compiler construction and formal methods [25, 24, 23].

4http://www.ist.tugraz.at/projects/ieac

5These functions are renamed with uppercase letters in order to distinguish them from the object language, these functions are operating on as part of the programming language. This avoids confusing the interpreter.
These three points shall contribute to the design of the CTP-based programming language. The above Pt.2 will be extended to an essential feature of future workplaces for engineering in Sect.3. Further details of the language are up to discussion and to re-design. Let us look at some of the details.

**An example program** written in the present language gives an algorithm solving a problem in structural engineering, which will serve again as an example in Sect.3.

01 Script bendingLine
02 (ł::real) (q::real) (ν::real) (b::real=>real) (rb::bool list) =
03 (LET
04 (funs:: bool list) =
05 (SubProblem (Bendingline,[bendingline,integrate],
06 [bendingline,integrate])
07 (real q, real b, real ν);
08 (equ::bool list) =
09 (SubProblem (Bendingline,[bendingline,setConstraints],
10 [bendingline,setConstraints])
11 (bools funs, bools rb, real l);
12 (sols::bool list) =
13 (SubProblem (Real,[equation,system,linear],[]))
14 (bools equ, reals [c,c₂,c₃,c₄]);
15 B = Take (LAST funs);
16 B = ((Substitute sols) @@
17 (RewriteAndInst [[bdv, ν], make_ratpoly_in]) B)
18 IN B)

The identifiers with ending underscores avoid type clashes with identifiers in the object language of formulas; the function constants real, bools, reals bring the arguments’ types into line for the list of arguments.

Most noticeable are the bulky function calls designated with *SubProblem*. These relate the descriptive and algorithmic aspect: the list of the functions’ arguments is preceded by a triple: #1 points to the respective theory (Bendingline or Real), #2 points to the (pattern of, see Pt.2 above) specification and #3 a method addressed by the same path (into the other tree of methods), because apparently there is only one method necessary for this problem.

Now let us look at the program code line by line:

**01.02** is the program header with the arguments. Note that the output-item \( b_\nu \) is also an argument, because all identifiers have been determined in the specification preceding the start of the program.

**03 LET** is as defined in Isabelle/HOL, which requires semicolons as delimiters except after the last line before IN in 18.

**04.07** designates a subproblem which #1 takes vocabulary from theory Bendingline, #2 relies on a specification addressed by \([\text{bendingline,} \text{integrate}]\) and #3 a method addressed by the same path (into the other tree of methods), because apparently there is only one method necessary for this problem.

**08..11** designates a similar subproblem (for the meaning of the program see Sect.3 p.33).

**12..14** calls a CAS function solving a system of equations. The method in #3 is empty, since the system is considered smart enough to find the appropriate algorithm in this case.

**15** takes the last element of the list funs \( \text{LAST} \) is renamed as part of the programming language in order to allow programs to operate on formulas with last without confusing the interpreter.

**16.17** shows forward chaining \( @@ \) of CAS functions, substitution and rewriting with a term rewriting system named make_ratpoly_in; rewriting is optimized for this univariate function with bound variable \( ν_\nu \).

There are immediately effective advantages from merging with the other lines of work:
Make all deductive components available within the programming language. So far, only matching and simplification can be accessed by the language primitives. In addition, decision procedures are useful for rewriting with conditional rewrite rules, provers could provide more powerful assistance in proving the preconditions of guards mentioned in Pt.2.

Further use of provers will be discussed in Sect.4.3.

Reuse Isar/ML/Scala integration: Presently Isabelle/Isar’s extra-logical infrastructure is evolving [39] towards open interoperability with front-ends. Some of the GUI front-ends under consideration for the Isar proof language are also appropriate as front-ends for programming and debugging. Also the communication between GUIs and the back-ends is similar in proof development and in program development. Thus Isabelle/Isar’s Scala API will serve both, proofs and programs, as shown in Figure 2.

The uniform architecture provides optimal prerequisites for an integrated workflow constructing algorithms and proofs in parallel. Since Scala runs on the JVM platform, the system is open for widespread use.

3 An engineer’s future workplace

All together, the work described above covers just a tiny part of the research contributing to an engineer’s future workplace. So many questions are open, that already strategic aspects are being addressed, for instance in [1].

However, we believe that a merge of the three lines of work will already establish notable advances without further research (respective questions open for research are discussed in Sect.4). With the motivation in mind to replace CAS-based programming with CTP-based programming, it seems appropriate to discuss these advances with respect to a practice oriented setting. We are keen enough to describe a future workplace of an engineer who programs some application for electrical engineering, mechanical engineering or the like.

A standard problem from a textbook on structural engineering might serve as an example for discussing details at the engineer’s future workplace:

*Determine the bending line of a beam of length L, which consists of homogeneous material, which is clamped on one side and which is under constant line load q₀; see Figure 3.*

*Hint: Use the constraints on the shear force \(V(0) = q₀ \cdot L\), on the bending moment \(M_b(L) = 0\) and on the expected bending line \(y(0) = 0\) and \(y'(0) = 0\).*

A specification of the problem in the traditional form with the input-items \(in\), the precondition \(pre\), the output-item(s) \(out\) and the postcondition \(post\) finally might look like this in notation used by engineers:
in : function \( q_0, \text{length} \ L \)

pre : \( q_0 \text{ is integrable in } x \land L > 0 \)

out : function \( y(x) \)

post : \( y(0) = 0 \land y'(0) = 0 \land V(0) = q_0 L \land M_b(L) = 0 \)

where \( V \) and \( M_b \) are constant function symbols in this theory of “bending lines”. \textit{function} and \textit{length} are functions fixing the arguments’ types; \( q_0 \) is a constant function with type \( \mathbb{R} \to \mathbb{R} \).

A program refining the specification has been listed already in Sect.2.3. The program’s algorithm is a straightforward refinement of the postcondition: the theory of bending lines is simple in this case, saying that the bending line’s \( y(x) \) fourth derivative \( y^{(4)}(x) \) is (almost) the load function \( q_0(x) \), \( y^{(4)}(x) = c \cdot q_0(x) \). Thus the first subproblem (program lines 04..07) integrates the load function \( q_0(x) \) with identifier \( f \) four times and returns for functions \( \text{funs} \) with four respective constants of integration, \( c, c_1, c_2, c_3 \).

The second subproblem (lines 08..11) substitutes the constraints, given by the program’s argument \( \text{rb} \) into the four functions \( \text{funs} \) and returns four equations \( \text{equs} \) containing \( c, c_1, c_2, c_3 \).

The third subproblem is a classical CAS problem solving the (linear, uniquely solvable) system of four equations in \( c, c_1, c_2, c_3 \) yielding the normal form of four equations in \( \text{sols} \).

Finally there are elementary CAS tasks, substituting the solutions \( \text{sols} \) into the last function from the list of four function \( \text{funs} \), which is the bending line \( y(x) \), and simplify this function.

Before entering description and discussion of the workplace we note, that the description necessarily is speculative.

In particular one prerequisite for efficient work at such a workplace, domain specific knowledge, is not yet present (and a discussion how to create such knowledge is out of scope of this paper).

Domain specific knowledge is supposed to undergo formalization and mechanisation more and more, in order to improve professionality of software development [4].

The differentiation of such knowledge is already prepared by the distinction between the two established notions, quality of design and quality of conformance. The former concerns the relation between “reality” and an abstract model, the latter the relation between the abstract model and an implementation. Both relations are increasingly investigated and tackled with mathematical methods. One of the strongest method is proof; thus one might predict increasing importance of mathematical proof in this part software technology.

Domain engineering is expected to create and mechanise domain specific knowledge [4], i.e. appropriate notions (for instance \textit{beam}, \textit{bending line} in the above example), concise abstractions like domain specific predicates (for instance \textit{homogeneous}, \textit{clamped} in the above example) and models, which will develop from kinds of abstract datatypes to algebraic structures with specific axioms, definitions, lemmas and theorems. So one might predict increasing importance of mathematical proof also in this part software technology.

Domain specific knowledge is already being at the beginning of mechanization in software engineering. It in Isabelle’s “Archive of Formal Proofs” there are collections like on “Computer Science >> Security” with growing lists like “SIFPL, VolpanoSmith, HotelKeyCards, RSAPSS, InformationFlowSlicing”, see \url{http://afp.sourceforge.net/topics.shtml}.

[5, 12, 7] describe some initial attempts.
In spite of the non-existence of such knowledge we are keen enough to predict some details, because these are already anticipated in the language as described in Sect.2.2:

Three aspects of knowledge are expected to be distinguished in the sequel; this distinction has not yet been established. For instance, Isabelle’s mechanisms for handling knowledge [38] are highly elaborated and show several levels of granularity: theories, local theories, locales, theory contexts, generic contexts, local contexts — all allow SML code inline to the code defining logical entities and anti-quotations within the inline SML code referring to the logical entities, and all packed into theories without explicit structuring so far.

There are initial ideas of how to structure knowledge [9]; here we just want to distinguish three general aspects of knowledge, the deductive, the applicative and the algorithmic aspect of knowledge:

1. **Deductive knowledge** is that kind of knowledge which traditionally is represented in so-called theories, for instance Isabelle’s theories.

2. **Applicative knowledge** is represented by specifications in the simplest case; for each method from Pt.3 below there should be a specification, but there might be more than one method refining one and the same specification. We shall call this kind of knowledge problems.

3. **Algorithmic knowledge** comprises specific algorithms solving problems specified in Pt.2 above; we shall call this kind of knowledge methods. This knowledge represents the bridge to algorithmic mathematics and classical programming.

Many questions about these three points are open, for instance, how to separate them, where to place respective parts of code. Also the structure seems only settled for theories (an directed acyclic graph), while the structure assembling methods and problems is open.

In order to make things more precise in the sequel, we simply assume methods and problems assembled in two separated trees.

Discussion of a futuristic workflow is speculative, as already mentioned. With respect to the above paragraph we expect the engineer being an software expert in the domain of structural engineering. And we can assume, that the problem of calculating the bending line is part of a larger problem, probably requiring the calculation of many bending lines for many structural components. What might this engineer do? We assume an iterative approach towards a final result using these steps:

Looking up the knowledge available at his or her electronic workplace will be the first step. The theories will be scanned, probably because some details have been forgotten (e.g. where the matter constants come in: \( y''(x) = -\frac{M_b(x)}{EI} \) (for the bending line \( y(x) \)), or because decisions are required whether to work with univariate or multivariate functions, whether to take a continuous or an approximating discrete model, etc.

The most interesting question will be, if there is already a solution for the given problem in the system; this requires searching problems (specifications) and algorithms (methods). If the input-items are already determined, a mechanical problem refinement is possible as described in Sect.2.3.Pt.2; if also a postcondition has been formulated, the search will be even more precise.

Let us assume, that there is still no appropriate solution in the system, and that the engineer decides to go towards the solution already presented in Sect.2.3.

Interactively assembling the parts of the code required will be the next step. First the problem will be specified — not for academic reasons, but because the solution for the problem will serve within a larger system, which requires specification for components. The formulation of the specification is supported by domain specific predicates (which has not yet been developed in our example).

According to the assumption, that no ready-made solution is available, we also may assume, that the first two subproblems (lines 04..07 and 08..11) are not available. So the engineer has to develop respective specifications.
and methods; probably this would be done iteratively; first the specification, then the algorithm; quick testing of preliminary code sequences in an input-response-loop is helpful, as already described in Sect.2.2.

For other parts solutions are ready-made, in our example this will be solving the system of equations (lines 12..14). In general, an appropriate method has to be selected from a variety of algorithms for systems of linear equations, depending on the system’s dimension etc.

**Proving correctness** of the program is not an additional exercise during or after development, rather it is enforced by the system — which poses a considerable challenge to the usability of that system: restrictive discipline must get counterbalance by noticeable advantages!

One advantage is mechanized code generation as mentioned in Sect.2.1. Another advantage is flexible support in assembling software components and decisive support in proving correctness of the development. Usually the number of proof obligations increases dramatically when tackling more complex problems. Even in our simple example lots of effort is required to prove that the program’s arguments and the results of subproblems fulfill the preconditions for a subsequent subproblem.

Further advantages will be proper handling of approximations by reals, i.e. by floating point numbers; Sect.2.2 already mentioned this issue, Sect.4.2 will discuss further ideas.

4 Research questions raised from merging

Recalling what has been said in the introduction, we do not claim to address the most essential issues in the design of CTP-based programming languages. We rather discuss in the sequel, how merging the three lines of work described in Sect.2 leads to novel chances for advancements. However, we claim that we address such advancements which are well into reach of R&D. And the aim remains a pragmatic one: a language which balances demands and benefits such that engineers like to use it for their programming tasks, preliminarily at least in engineering education.

4.1 Traceability of mathematics applications

Engineers are responsible for what they deliver; this is an issue with respect to the growing complexity of the tools engineers use at their workplace. With respect to software tools specifications improve manageability, as already discussed. Another means is, to make software transparent such that the engineer has the chance to trace what is going on internally; this is an additional support for taking over responsibility in engineering.

Traceability poses research issues, some of which are already in reach to be accomplished by available concepts and technologies:

**Manage partiality conditions** such that an engineer can trace down where singularities come from, where solutions of equations might have disappeared etc.

Sect.2.2 presented what already has been achieved for CAS functions if evaluated in an input-response-loop. Embedding CAS functions in arbitrary complex software reinforces the issue for the design of respective programming languages.

Looking at the concept of “context” and the respective mechanisms, Isabelle/Isar seems to provide much of the logical infrastructure necessary to cope with handling partiality conditions dynamically during program execution: The context has to be initialized with the input-items and the preconditions of the specification at beginning the execution of a function; in case a partial function causes some additional assumption, this assumption can be added to the context. The dynamic scoping of contexts can follow the block structure of nested function calls.

Detailed design of the language has to consider means to make the dynamic expansion of contexts traceable in engineering practice. Nearby possibilities for such means are extensions of debuggers, another means could be dedicated tracing facilities.
Interactive decomposition of terms is another approach to improve traceability of programs which apply mathematics; this approach exploits the power of type systems.

Many proof assistants already include parsing mechanisms that allow type inference and disambiguation of overloaded notations. This mechanism can not only be adapted to allow the user to type mathematical formulas in a CAS way, but also to trace the actual definitions of symbols in an expression. In a typical interaction with a proof assistant, definitions and properties have to be given explicit types unless the types can be automatically inferred. This can be done in many cases. For example given the expression $2 \ast \pi \ast r$, a proof assistant knows that $\pi$ is a real number. Then type inference and overloading can infer that $r$ is a real number free variable, that $2$ is a real constant and $\ast$ is the real number multiplication. Also proof assistants have mechanisms for finding out definitions of the given symbols. In the example above it is possible to find the associated definition of $\pi$ and real number multiplication.

So, the principle seems clear: a typed term contains all information necessary to ensure correct manipulations in programs, and this information shall be made available to the user — in principle. However, the question is open, how to make this information accessible in interaction on formulas, such that it is really useful in engineering practice.

### 4.2 Two steps in a never ending story

Two of the most challenging problems in making mathematics software safe and reliable are multivaluedness of CAS-functions and the handling of floating-point number constants (in principle indefinite, if of type real) in (finite!) computers. Since these problems are urgent as well, even little steps towards accomplishment like the following ones might be valuable.

**Multivaluedness**: The main motivation for developing the certified CAS prototype (described in Sect.2.2) was to provide the users with a system that is guaranteed to never make mistakes. One source of mistakes is multivaluedness; systems still get confused between branches of ‘multi-valued’ functions [16].

The description of the certified CAS prototype also shows a mistake that Maple makes when dealing with a complex function with multiple branches, however the certified system is not able to deal with it formally at all. A first workaround may be to define multivalued functions in a similar way to the partial functions [17]. Such approach would guarantee no mistakes, but would allow only computations which reside entirely within one branch of a multivalued function. For performing computations that cross branches, a formalization of Riemann surfaces is necessary. The libraries of the proof assistants contain very few theorems that talk about multi-valued functions, so such a theory needs to be developed together with certified decision procedures for this domain.

**Real numbers** are one of the basic features of a system for an engineer. Therefore major CAS systems include efficient algorithms for dealing with real number computations and approximations. Numeric methods (e.g. interval arithmetic) are used to provide correctness of these approximations. Many proof assistants include formalizations of real numbers that provide mechanisms for computing approximations. The certified CAS prototype described in Sect. 2.2 uses the Boehm-style calculations already present in HOL Light standard library. Unfortunately, the approximations done by inferences are quite inefficient.

O’Connor’s work [33] allows working with infinite precision real numbers in Coq effectively. The proofs can be used for computation inside the proof assistant (with the help of reflection and the Coq bytecode engine), as well as in the extracted programs. Additionally the programs extracted from constructive proofs are naturally functional and therefore can be easily parallelized, which makes them even more efficient. However the approach requires constructive proofs. It is possible to extract proof from classical proofs (e.g. with A-translation), but this has not yet been experimented with in the setting of real numbers.

### 4.3 Generalization of manipulated objects

This paper assumes the development of programs comprising formal specification, coding algorithms and proving properties of programs — all these development activities operate on formulas: but that does not mean, that also programs operate on formulas. This is not even true for mathematics: mathematical theories comprise, besides
formulas, geometric objects like points, circles, straight lines and the like; they comprise graphs of various kinds, and probably several other kinds of objects.

So, when considering design issues of a CTP-based programming language, considerations about mathematical domains beyond predicate calculus seem appropriate. Here the domain of geometry is considered.

Formalizing geometry theorems in proof assistants is quite challenging. As shown in [26] the non-degeneracy conditions and the amount of incidence relations induce many technical lemmas and side conditions leading to technical proofs. This is why quite often special approaches are used in particular formalizations.

On the other hand, the language of a CAS can be used to talk about geometric constructions. The transformation of properties of the constructions with the help of symbolic computation and the general algorithms present in a system (Cylindrical algebraic decomposition, Gröbner bases) allows showing properties of those constructions even in non-standard geometric relations [15]. Specifying such constructions in a computer-algebra like language with strong semantics would allow building real computer verified proofs of properties entailed by geometric constructions.

So, if considering the design of a CTP-based programming language, it seems worth the effort to extend the scope of the language; the relation between deduction and programming comes into a broader view, which might well lead to novel details in design.

5 Related work

The union of the three lines of work merged in this paper relates to a wide range of research topics; so we select only the most relevant work.

Languages of proof assistants are different from the language envisaged, and we did not tackle the question of integrating the different kinds of languages. In particular in the case of Isabelle, the proof language is Isar [31]. Isar is presented to end-users as a language for human-readable proofs, i.e. proofs as close as possible to traditional notation for mathematical proofs. Algorithmic language elements are accessible, but appear as an impurity (imagine Sect.2.3 programming bending lines in Isar).

Since languages of proof assistants have algorithmic language elements, and the envisaged language also comprises deductive elements, the question about integration arises; this questions seems very interesting, but is out of scope of this paper.

Integration of deduction and programming has found a mathematical foundation in the refinement calculus [2], a theory of refining specifications to programs. The same author develops software supporting this kind of refinement [3] for imperative programs; however, for programming applications in mathematics we prefer functional programming, which concerns a subset of the above theory.

The Focalize system [7] is close to the language proposed here. Much can be learned, in design details and in usability, from “design by contract” [28] and respective systems [8], from work done in the Kestrel Institute [8], from the B-Method [9]. Another system combining specification, proof and other engineering tasks is PVS [34]. Albeit not being a typed system, also ACL2 [10] provides integration of deduction and programming.

Computer Algebra and Proof Assistants have coexisted for a many years so there is much research trying to bridge the gap between these approaches from both sides.

First, many proof assistants already have CAS-like functionality, especially for domains like arithmetic. They provide the user with conversions, tactics or decision procedures that solve problems in a particular domain. Such decision
procedures present in the standard library of HOL Light are used inside the prototype described in Sect.2.2 for arithmetic’s, symbolic differentiation and others.

Similarly some CAS systems provide environments that allow logical reasoning and proving properties within the system. Such environments are provided either as logical extensions (e.g. [37]) or are implemented within a CAS using its language [6].

There are numerous architectures for information exchange between CAS and CTP with different levels of degree of trust between the prover and the CAS. In certain approaches the prover uses the algorithms present in the CAS without checking their correctness, i.e. as an oracle, whereas in other approaches the prover takes the output of a CAS and then tries to build a verified theorem out of it. A longer list of frameworks for information exchange and bridges between systems can be found in [18].

There are many approaches to defining partial functions in proof assistants. Since we would like the user to define functions without being exposed to the underlying logic of the proof assistant we only mention some automated mechanisms for defining partial functions in the logic of a proof assistant. Krauss [20] has developed a framework for defining partial recursive functions in Isabelle/HOL, which formally proves termination by searching for lexicographic combinations of size measures. Farmer [10] implements a scheme for defining partial recursive functions in IMPS.

Real Number theories together with accompanying decision procedures already exist in many proof assistants. Melquiond has created a Coq tactic that can solve some linear inequalities over real number expressions using interval arithmetic and bisection [27]. Obua developed a computing library for Isabelle, which uses computation rather than deduction to obtain bounds on real number expressions. HOL Light constructed real numbers are described in [13]. A mechanism for approximation of real number present in the standard library uses the fact that HOL Light terms are transparent and decomposes a term or goal into subterms, looking for underlying real number operations or constants. Lester implemented approximation of real number expressions in PVS [22], which uses fast converging Cauchy sequences to obtain effective evaluation inside PVS.

6 Summary and future work

This paper was set off by a simple question: What about proceeding from the successful CAS-based programming languages to a more powerful basis provided by CTP? The question immediately makes clear that answers are embedded in an apparently never ending (but still topical) story, in the integration of deduction and computation. In order to not get lost in this story we choose a concrete and pragmatic approach.

Our approach is concrete in that it starts design considerations from three lines of work pursued independently so far: (1) narrowing the gap between declarative program specification and program generation already working in Isabelle, (2) reusing work on a CAS-like input-response-loop embedded into HOL Light, and (3) reconstructing an experimental language for applied mathematics based on Isabelle/HOL. Sect.2 presents these three lines together with immediate advantages from respective merges: code generation and relief from typing problems in programming on complex mathematical structures, handling partiality of CAS-functions as a sound basis for execution of mathematics applications in programs, availability of provers in a programming language checking the conditions in guards and reuse of Isabelle’s ML-Scala integration.

Sect.4 deals with advantages of merging the three lines of work, which are not so obvious but definitely within reach of actual R&D. Sect.4.1 argues that specifications guarding programs enhances soundness of implementation; but this seems not sufficient: in order to support interactivity in integrating work on specifications and on algorithms, novel means for tracing system behavior are required. Tracing down details in type-definitions into (sub-)terms and tracing partiality conditions are discussed. Sect.4.2 proposes steps towards accomplishing two challenging problems for mathematics software, multivaluedness of CAS-functions and floating point numbers of type real. Sect.4.3 tries to open up the scope for the design of CAS-based programming languages, and also discusses some specific requirements for constructive geometry.

Our approach is pragmatic in that it envisages design decisions towards usefulness in engineering practice. Sect.3 discusses two activities at an engineer’s electronic workbench, which seem indispensable in the future: domain engi-
neering and proving. For domain specific knowledge the distinction of three aspects is suggested: the deductive, the applicative and the algorithmic aspect.

The authors are aware, that only prototyping can explore usability of the features. Nevertheless the authors are convinced that in the novel features of CTP-based programming languages can balance demands and benefits such that such systems will be used in the future.

Future work, if going towards more detailed design of a programming language and towards an experimental implementation, will require expertise from the disciplines of compiler construction and of symbolic computation. A decision has to be made whether the features of SML (module system, abstract datatypes and respective pattern matching, exception handling) should be lifted into a language in Isabelle/HOL (like the experimental language described in Sect.2.3), or whether the high-level constructs of Isabelle/HOL can be traced/pushed down to the implementation language SML. In the latter case traceability (Sect.4.1) could emerge from adaption of the SML debugger.

References


Generating certified and efficient numerical codes requires information ranging from the mathematical level to the representation of numbers. Even though the mathematical semantics can be expressed using the content part of MathML, this language does not encompass the implementation on computers. Indeed various arithmetics may be involved, like floating-point or fixed-point, in fixed precision or arbitrary precision, and current tools do not handle all of these.

Therefore we propose in this paper LEMA (Langage pour les Expressions Mathématiques Annotées), a descriptive language based on MathML with additional expressiveness. LEMA will be used during the automatic generation of certified numerical codes. Such a generation process typically involves several steps, and LEMA would thus act as a glue to represent and store the information at every stage.

First, we specify in the language the characteristics of the arithmetic as described in the IEEE 754 floating-point standard: formats, exceptions, rounding modes. This can be generalized to other arithmetics. Such a generation process typically involves several steps, and LEMA would thus act as a glue to represent and store the information at every stage.

Next, we use annotations to attach a specific arithmetic context to an expression tree. Finally, considering the evaluation of the expression in this context allows us to deduce several properties on the result, like being exact or being an exception. Other useful properties include numerical ranges and error bounds.

1 Introduction

A major problem with numerical applications (from libraries to user code) is the control of accuracy with acceptable performance. While some applications are more focused on performance, for other ones, accuracy of the results is crucial. For instance, the requirement can be:

- a guarantee on an error bound specified by the user;


### LEMA: Towards a Language for Reliable Arithmetic

Vincent Lefèvre\(^a\) Philippe Théveny\(^a\) Florent de Dinechin\(^a\) Claude-Pierre Jeannerod\(^a\) Christophe Mouilleron\(^a\) David Pfannholzer\(^a\) Nathalie Revol\(^a\)

\(^a\)UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL, Université de Lyon, \{vincent.lefevre, philippe.theveny, florent.de.dinechin, claude-pierre.jeannerod, christophe.mouilleron, david.pfannholzer, nathalie.revol\}@ens-lyon.fr. This work is supported by the ANR project EVA-Flo.

Abstract

Generating certified and efficient numerical codes requires information ranging from the mathematical level to the representation of numbers. Even though the mathematical semantics can be expressed using the content part of MathML, this language does not encompass the implementation on computers. Indeed various arithmetics may be involved, like floating-point or fixed-point, in fixed precision or arbitrary precision, and current tools do not handle all of these.

Therefore we propose in this paper LEMA (Langage pour les Expressions Mathématiques Annotées), a descriptive language based on MathML with additional expressiveness. LEMA will be used during the automatic generation of certified numerical codes. Such a generation process typically involves several steps, and LEMA would thus act as a glue to represent and store the information at every stage.

First, we specify in the language the characteristics of the arithmetic as described in the IEEE 754 floating-point standard: formats, exceptions, rounding modes. This can be generalized to other arithmetics. Then, we use annotations to attach a specific arithmetic context to an expression tree. Finally, considering the evaluation of the expression in this context allows us to deduce several properties on the result, like being exact or being an exception. Other useful properties include numerical ranges and error bounds.
• correct rounding to some given target format, that is, the result obtained as if all internal computations were carried out in infinite precision before the final rounding;
• an exact result, when it is known to be exactly representable in some format.

In addition to accuracy, the range of the computed values needs to be considered too, in order to either avoid overflows and underflows, or control what happens if such exceptions occur. Arithmetics for which this kind of properties can be guaranteed are referred to as reliable arithmetics.

Even applications focused on performance need meaningful results, which one may want to certify.

Values (either final or internal results) can be expressed in various arithmetics, which can be mixed even within a common application for, say, performance reasons. The most common ones are (see [1] and [24] for comprehensive descriptions):

• integers, either bounded (in a fixed format) or in arbitrary precision;
• floating point in some radix $\beta$ (in general, $\beta = 2$ or 10) and precision $p$ (fixed or arbitrary): a number $x$ has the form $x = s \cdot m \cdot \beta^e$, where $s = \pm 1$ is the sign, $m = x_0.x_1x_2 \ldots x_{p-1}$ (with $0 \leq x_i \leq \beta - 1$) is the significand, and the integer $e$ is the exponent (whose range can be bounded or not);
• fixed point, which is quite similar to integer arithmetic.

But one can also construct derived arithmetics, such as double-double arithmetic [23, 9], where each number $x$ is represented by a pair of double-precision floating-point numbers $(x_{hi}, x_{lo})$ and interpreted as $x = x_{hi} + x_{lo}$ (exactly).

The variety of architectures or inputs implies the support of some previously mentioned arithmetics, which in turn yields many variants in the implementation of a single algorithm. In order to avoid tedious work for similar solutions, we seek to generate efficient code automatically.

However, speed is not our main concern. We also want to use the same description of inputs and of the chosen arithmetic so as to prove the validity and accuracy of the implementation with formal methods. Indeed, a traditional solution is to validate the programs with hand-written proofs. In the domain of numerical applications, such proofs often require many tedious calculations, and in practice, they often contain errors (e.g., in corner cases), making them globally incorrect. Moreover if the problem slightly changes (due to a different hypothesis, architecture, input...), the whole proof may need to be manually checked or even redone.

To satisfy these needs, we present here a new descriptive language called LEMA.

The outline of this paper is as follows. We start in Section 2 by presenting several established software environments. We show there why they do not allow us to achieve directly the goals mentioned above, and how the single language LEMA could act as a glue between some of them. After describing the LEMA language requirements in Section 3, we argue our decision to use XML as a basis for LEMA in Section 4. Then Section 5 illustrates with the help of examples the use we make of the tree nature of XML and its extensibility. In particular, we show how mathematical expressions are annotated with their evaluation in a given floating-point context and with associated proofs.

2 Related Work

2.1 Existing Environments to Develop Numerical Codes

Various environments already exist for either generating efficient numerical codes or analyzing/proving the numerical properties of given programs. But, like any software, such tools have their own limitations. One can cite:

• Spiral\(^1\) This project develops a program generation system capable of producing C or Verilog code implementing digital signal processing algorithms in (complex or real) floating-point arithmetic, like DFTs [27, 28], and more recently, some non-linear transforms as well [15]. Given some architectural features (like, typically, various forms of parallelism: SMP, SIMD, etc.), the Spiral system automatically optimizes codes for efficiency. However, although overflows are avoided in some specific applications as described in [8], the generation process does not seem to implement error-analysis techniques.

\(^1\)http://spiral.net/
• **Fluctuat.** This software is a static analyzer devoted to the study of the propagation of rounding errors inherent to floating-point computations [29, 16, 10]. Given a (possibly very large) numerical code, Fluctuat can detect automatically catastrophic losses of accuracy and identify the corresponding faulty variable(s). Also, guaranteed error bounds are computed using affine arithmetic [2, 7]. However formal proofs are not produced and since Fluctuat is not freely available, its extension is an issue.

• **Why.** This platform for code certification is based on the idea that properties of some existing code can be proven by annotating the code through comments. Then these comments are turned into different proof obligations. Afterwards, one can deal with each proof obligation using an appropriate tool. Frama-C, a C source static analyzer, has a plug-in named Jessie which converts C code with comments written in the ANSI/ISO C Specification Language (ACSL) into Why code, thus allowing to reuse all the features of the Why platform.

The main limitation of this approach lies in its restriction to one particular programming language. Frama-C for instance only works on preprocessed C code and will not yield a proof for some extended class of C implementations. Moreover, working on a specific code, rather than at the mathematical level, has two drawbacks. First, useful information could have been lost when the developer has implemented his algorithm. Second, it is conflicting with our idea to go from a mathematical description of a problem to code generation.

### 2.2 Tools Commonly Used for Specific Tasks

On the other hand, to assist the design of fast and provably-accurate implementations of mathematical functions, the following tools can be extremely useful. However, as we shall see, each of them is used to perform very specific tasks, so that the programmer usually has to combine them in order to get a full design flow:

• *Computer algebra systems such as Maple.* Symbolic computation can be useful to simplify symbolic expressions or to obtain mathematical properties like the monotonicity of a given function.

• **Sollya.** The implementation of mathematical functions is often performed by means of polynomial approximation. Sollya is able to compute such an approximation. In addition, one can ask for a guaranteed upper bound on the supremum norm of the difference between a function and its polynomial approximant, thus allowing to control the error committed at this approximation step.

• **Gappa,** “a tool intended to help verifying and formally proving properties on numerical programs dealing with floating-point or fixed-point arithmetic.”

• **CGPE.** This tool returns some efficient and certified evaluation schemes for univariate and bivariate polynomials, optimized for a specific target architecture. Even though code latency is the main focus, one can ask that the error due to the evaluation order does not exceed a given bound.

• **Tools to search for “worst cases”.** [19] Such tools target specifically the correct rounding of mathematical functions in a fixed precision, mainly double precision. From the input data (precision, function, tested interval, etc.), these tools generate C code that performs the search. The most important part related to code generation is here a hierarchical approximation of a high-degree polynomial by degree-2 polynomials on consecutive sub-intervals, using a modulo-1 fixed-point arithmetic (easily emulated with the mpn layer of GMP), where each variable has its own precision (which must be a multiple of 32 or 64 bits, depending on the target). Even though dynamical error analysis is done to automatically determine the precision of each variable, all the proofs have been done manually. Some optimizations for current architectures or for specific functions would need to redo most of the error analysis and modify a large part of the tools. This problem was actually at the origin of LEfMA, and its need to support some exotic arithmetics.

---


3. Still, academics can ask Fluctuat’s developers to obtain it.


7. http://www.maplesoft.com/Products/Maple/


9. http://gappa.gforge.inria.fr/ as well as [22], [6], [4].

- GNU MPFR\textsuperscript{11}. This library offers floating-point arithmetic in arbitrary precision with correct rounding. Here, arbitrary precision means that the precision is a parameter of each function call. Note that this library underlies the computations of the software tools mentioned in this section.

Most of these tools have been routinely used to produce and validate significant parts of the codes of mathematical software libraries like CRlibm\textsuperscript{3} and FLIP\textsuperscript{13}.

### 2.3 Tool Integration Through LEMA

When implementing an application with certified numerical properties, one can in turn use some of the tools that have been previously presented. Note that they do not act on the same type of data: the information needed by a proof assistant may not be relevant to a general purpose computer algebra system, and a polynomial specialized tool may not be able to deal with abstract mathematical properties. As there certainly is a natural sequence of calling these tools from the mathematical level to the hardware one, we could have inserted small translators between each application in order to automate the whole process. But some steps would also need data of the specification not produced by its predecessor. For instance, the hardware instruction set and the characteristics of each useful instruction are needed when generating efficient code but not before. Indeed, it is unlikely that a computer algebra system would process them. Conversely, knowledge of the abstract level may be relevant at any time: the mathematical properties often help to choose the most efficient algorithm at the implementation stage. For this reason, we want to gather information produced by all the tools and to store them along with the specifications of the problem.

Figure 1 shows how we intend, in the long term, to address the issue of certified numerical code generation using LEMA: The user provides a description of a problem written in LEMA; then this description is enriched by interaction with the previously mentioned tools; finally, when enough information has been gathered, the user can proceed with the generation of code, possibly with proof.

Generated code can be in C language, possibly with function calls to some libraries, such as GNU MPFR if multiple precision is needed. Other targets are possible, such as the GCC middle-end, VHDL, or a proof assistant language. Interactions with these tools (represented as arrows in Figure 1) use their native script languages.

### 3 Requirements Analysis

We analyze in this section the requirements of a language that must be sufficiently rich to describe numerical algorithms without loss of information.

We could have reused the annotation languages (like ACSL) that have been presented in the previous section but they are quite general and have only limited support for floating-point arithmetic. Instead of extending one of them, we propose a single language for all data. Such data are either problem specifications provided by the user, or properties generated by some tools (e.g., by a tool doing value range propagation or error analysis); they can also be some information that does not belong to the specifications. The latter case includes hints for the proof assistant provided by the user because of the difficulty of finding them in an automatic way. The language should hold the status of these data: how they have been added, whether they need to be checked by some prover, whether and how they have been checked, and so on.

A consequence of integrating all kind of information in one place is that this language must mix different domains: abstract mathematics, floating-point arithmetic, and hardware capacities. In each domain, we want to express as simply as possible some basic and high level properties.

In the mathematical domain, it shall be possible to express any kind of mathematical expression or relation that a computer algebra system can understand. The way expressions are written must preserve their intrinsic parallelism, that is, their independence with respect to an order of evaluation. This is another reason to exclude annotations in a procedural language.

\textsuperscript{11}http://www.mpfr.org/ and \textsuperscript{14}. 
In the floating-point domain, we want to express simple properties like exactness in a computation, error bound of an evaluation, or the range of a floating-point function, and there shall be possible to bind such properties to their proofs. To state initial specifications, we need a higher level of expressiveness to describe

- various arithmetics, including arbitrary precision, but also particular arithmetics such as fixed point modulo 1 in arbitrary precision (needed to generalize the hierarchical approximations of a function by polynomials, as described in [19]);
- the rounding mode and its properties;
- the measure of error as a function of the exact value and its approximation;
- how standards like IEEE 754-2008 [17] and C99 [18] define the behavior of functions on special values;
- formal proofs, possibly with properties declared as hypotheses (or axioms) if it is too difficult or impossible to prove them with current tools.

In the hardware description domain, the useful information ranges from which integer and floating-point formats are available on the target platform, to a more complex level for the description of the instruction set characteristics in terms of execution time and parallelism capabilities.

And most of all, the language must provide innate extensibility, so that we can create it in a minimal form and enhance it progressively in response to our needs.

4 Rationale for XML and MathML

The first thing we need to express is numerical expressions. For the moment, let us ignore problems related to rounding. The problem that needs to be solved is generally expressed in a mathematical form, and we want to remain close to this form. This is better for the understanding of the computations and for the proof. For this reason, imperative style such as in C or FORTRAN with variables that can be reused in control structures such as loops was rejected. A functional style is highly preferable. We considered using XML as a basis for our language, since
• it allows us to define the features we want, including clean extensibility (e.g., via namespaces);
• it natively expresses trees, thus perfectly fits to the functional style;
• the use of decorations permits the representation of properties parametrized by the arithmetic, without duplication (see Figure 2 for an example):

![LEMA Diagram]

Figure 2: LEMA tree with properties depending on arithmetics.

• it already has various validating parsers, making this solution rather robust (validation could be seen as the first step of a proof: we ensure that what the user writes has an unambiguous meaning, and the various tools working on this language will always get a sane input).

Then the question is: how do we write numerical expressions, which are in fact mathematical expressions? We could define our own XML elements for that, but since specification work has already been done with the XML-based language MathML, we choose to reuse it and, in particular, the content markups. As a bonus, this allows to interoperate with other tools that understand MathML, though this is not the primary motivation.

Nevertheless, the content markup is not intended to express floating-point concepts and we need to extend it for this purpose. As we will see in the following section, we choose to extend MathML using the XML syntactical extensions. An alternative design would have been to define new symbols in OpenMath content directories [26], but it leads to heavier and less manageable documents.

5 Annotating MathML Expressions With Evaluations

In this section, we present how we have extended the MathML language in order to support floating-point evaluations. As a simple example, let us evaluate the interval

\[ [0.17, 1071481169606510337534739638811517442326528] \]

in single precision (that is, in the binary32 format defined in the IEEE 754-2008 standard [17]) with rounding to nearest mode and even-rounding rule for the halfway cases. The exact mathematical interval we choose can be written in pure MathML as follows:

```xml
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <interval>
    <cn id="left" type="real">0.17</cn>
    <cn id= "right" type="integer">
      1071481169606510337534739638811517442326528
    </cn>
  </interval>
</math>
```
Here, both endpoints are numbers for briefness, but they could be more sophisticated expressions as well, like polynomials or rational functions. Using a tool like Gappa, we can produce certified evaluations in any floating-point context and then include this information in the previous document. From a lexical point of view, what we want to provide in XML consists of

- means to bind evaluated values to the exact value from which they stem,
- means to record floating-point properties of these evaluations,
- means to attach certificates to evaluations.

From an efficiency point of view, we want each expression to be evaluated only once in a given floating-point context. Consequently, evaluated values have to be easy to find knowing the exact value node, and vice versa. Additionally, evaluated values must be recorded in such a way that they can be sent to external tools in text form with minimal manipulation.

### 5.1 Numbers in MathML

First, let us review the support of numbers defined in the content markup section of MathML 3.0 (see §4.2.1 “Numbers \(<\text{cn}>\) in [20]). As we can see in the above example, the content number element \(<\text{cn}>\) encodes numbers and it specifies their kind with the \text{type} attribute. In strict MathML, the \text{type} attribute may only take four values: \text{integer}, \text{real}, \text{double}, and \text{hexdouble}, the last two ones being dedicated to floating-point numbers in double precision. This restriction to a unique precision prevents us to use them as a general means for floating-point number encoding. Numbers of the \text{real} type are written as “an optional sign (+ or –) followed by a string of digits possibly separated into an integer and a fractional part by a decimal point. Some examples are 0.3, 1, and –31.56.” The drawback with this type is the absence of exponent, which hinders its use for numbers whose magnitude is either tiny or huge. In non-strict MathML, floating-point numbers can be encoded with the \text{e-notation} type. For instance, \(<\text{cn type="e-notation">12.3<\text{sep/>3</cn}>\) represents 12.3 times 10³. Note also that in non-strict MathML a \text{base} attribute can be used to specify the base in which the text content of the \text{cn} element should be interpreted. This \text{e-notation} type allows writing floating-point numbers in arbitrary precision but it requires a conversion before the number can be read by non MathML-aware tools.

Furthermore, the MathML vocabulary can be enriched using symbols defined in OpenMath content dictionaries and, indeed, the content dictionary bigfloat1.cd of [5] defines a general floating-point representation of numbers. For example, the following sample encodes the floating-point value 12.625 = 101 · 2⁻³:

\[
\begin{align*}
\text{apply} & \\
\text{csymbol cd="bigfloat1">bigfloat</csymbol> & \\
\text{cn type="integer">101</cn> & \\
\text{cn type="integer">2</cn> & \\
\text{cn type="integer">3</cn>
\end{align*}
\]

Again, a number encoded this way has to be converted before being sent in text form to external tools.

### 5.2 Floating-Point Numbers in LEMA

To address the problem of consecutive conversions, we choose a floating-point number representation stricter than the one specified in the IEEE-754 standard (see §5.12.3 “External hexadecimal-significand character sequences representing finite numbers” in [17]): first, the mandatory sign, followed by the ‘0x’ prefix, and the hexadecimal integral significand with digits in lower case; second, the binary exponent in the following form: the exponent indicator ‘p’, the mandatory exponent sign, and at last the exponent value written in decimal. For instance, the right endpoint value in our example

\[
1071481116960651033753479638811517442326528
\]

is written in this format as

\[
+0x7bp+136.
\]
This allows us to represent binary floating-point numbers of any precision. It will be compatible with any tool that reads floating-point inputs with the C standard library, or multiple precision inputs with the GNU MPFR library.

Since such an encoding is not available in MathML, we define in a new namespace the special attribute `lema:type` of the `cn` element. The values of `lema:type` are custom floating-point types like “Binary32” for a floating-point number belonging to the set defined by the binary32 format of the IEEE 754-2008 standard [17] and they indicate that the text content of `cn` should be interpreted as a number written in the form described above.

The same LEMA namespace is also used for any other floating-point properties not already defined in MathML. We define some additional attributes for the `cn` element: the floating-point format and the rounding mode are specified in separate attributes called `lema:type` and `lema:rounding`, respectively. This helps filter the document for a particular precision or rounding mode. Moreover, each of the values `lema:type` and `lema:rounding` is used as a suffix to the value of the initial `id` attribute in the evaluated number `id`, so that the connection between the initial value and its evaluation is immediate to a human reader. This entails some data duplication but consistency is easy to check. Finally, the boolean attributes `lema:exact` and `lema:overflow` indicate, respectively, if the evaluation is exact\(^\text{12}\) and if it overflows. The example may be rewritten as follows:

\[
<cn id="right_Binary32_Nearest"
  lema:type="Binary32"
  lema:rounding="Nearest"
  lema:exact="true"
  lema:overflow="true">+0x7bp+136</cn>
\]

5.3 Annotating with Floating-Point Evaluations

Whereas we had to extend the MathML number encoding, we can reuse the mechanism provided by MathML to annotate elements with application specific information: the pair `<semantics>, <annotation-xml>` of elements (see §4.2.8 “Attribution via semantics” in [20]).

The `semantics` element is a container whose first child is the expression being annotated and whose other children are the annotations, each annotation being enclosed in an `annotation` or `annotation-xml` element. By this means, it is possible to provide several alternative presentations of the expression or to change its mathematical meaning with additional information. We use this annotation system to attach an evaluated value to the exact value from which it is derived. This evaluated value can be seen as an interpretation of the mathematical value in a given floating-point context. The `encoding` attribute of an `annotation` element indicates the data format of its text contents; for our needs we use the `application/lema-evaluation+xml` value, thus following the OpenMath example which uses `application/openmath+xml` and the recommendations of RFC 3023 for XML Media Types [25]. Therefore, our example with the evaluations can be written as follows:

\[
<math xmlns="http://www.w3.org/1998/Math/MathML"
  xmlns:lema="http://www.ens-lyon.fr/LIP/Arenaire/lema">
  <interval>
    <semantics>
      <cn id="left" type="real">0.17</cn>
      <annotation-xml lema:type="Binary32_Nearest"
        encoding="application/lema-evaluation+xml">
        <cn id="left_Binary32_Nearest"
          lema:type="Binary32"
          lema:rounding="Nearest"
          lema:exact="false">+0xae147bp-26</cn>
      </annotation-xml>
    </semantics>
    <semantics>
      <cn id="right" type="integer">12</cn>
    </semantics>
  </interval>
</math>
\]

\(^{12}\)The notion of exactness is defined regardless of the range of the exponent.
As a \textit{semantics} element admits several \textit{annotation-xml} children, we can attach to a single number many evaluations in different floating-point contexts. Contexts are differentiated with the \textit{lema:type} attribute of the \textit{annotation-xml} element, which eases the retrieval of a given evaluation by just browsing among siblings of the exact number node. Thanks to this proximity, the converse operation, that is, finding the exact value knowing the evaluated value node, is simple too.

5.4 Annotating Evaluations with Certificates

Furthermore, we would like to certify, typically by using Gappa, evaluated values and properties such as exactness. In the following example, we present an excerpt where such a Gappa proof is embedded after the evaluated value using the ability of the \textit{annotation-xml} element to contain application-specific elements:

\begin{verbatim}
<semantics>
  <cn id="right" type="integer">
    10714811169606510337534739638811517442326528
  </cn>
  <annotation-xml lema:type="Binary32_Nearest"
    encoding="application/lema-evaluation+xml">
    <cn id="right_Binary32_Nearest"
      lema:type="Binary32"
      lema:rounding="Nearest"
      lema:exact="true"
      lema:overflow="true">+0x7bp+136</cn>
    <lema:proof href="right_Binary32_Nearest"
      type="gappa">
      <![CDATA[
@rndn = float<24, -126, ne>;
MaxFloat = 0xf.fffffp+124;
right = 10714811169606510337534739638811517442326528;
right_Binary32_Nearest = +0x7bp+136;

{ right_Binary32_Nearest - rndn(right) in [0, 0]
  \land right_Binary32_Nearest - right in [0, 0]
  \land right_Binary32_Nearest - MaxFloat >= 0 }
]]>
    </lema:proof>
  </annotation-xml>
</semantics>
\end{verbatim}
Here we have introduced a new `lema:proof` element as a container for the script proving the evaluation whose identifier is referenced by the `href` attribute. Several proofs for the same evaluation but written in different tool languages can be embedded at this point, and the `type` attribute differentiates them. Here, the CDATA content is a script that Gappa can interpret. A Gappa script is composed of two or three sections: The first one is where symbols are defined; the second one, written between curly brackets, is a logical formula to be proven; the last one, which is optional, is a series of hints like rewriting rules or bisection directives, and is aimed at helping the Gappa engine (see [21] for further information).

In the above example, the first four lines in the CDATA part belong to the definition section: the symbol `rndn` defines both the rounding mode and the floating-point format, while `MaxFloat` corresponds to the maximal number that can be represented in the binary32 format. The values of the `id` attributes are reused to define symbols for the corresponding numbers, making it easier to map them to their counterpart in the XML document.

The lines between curly brackets form the logical formula, which is a conjunction of statements. The first line states that the rounded value truly is `+0x7bp+136`, the second one, that the exact value of the right endpoint is represented without rounding error, and the last one that it actually overflows.

From this script, Gappa can also derive formal proofs in Coq and HOL Light, which could be embedded next to it. But as the formal proofs are mere certificates that are not used thereafter and that should not be corrupted by subsequent transformations, it is preferred to save them in external files, using the `src` attribute value to record their URI, as in:

```xml
<semantics>
  <cn id="left">0.17</cn>
  <annotation-xml lema:type="Binary32_Nearest" encoding="application/lema-evaluation+xml">
    <cn id="left_Binary32_Nearest"
      lema:type="Binary32"
      lema:rounding="Nearest"
      lema:exact="false">
      +0xae147bp-26
    </cn>
    <lema:proof href="left_Binary32_Nearest" type="gappa"
      src="left_Binary32_Nearest.gappa"/>
    <lema:proof href="left_Binary32_Nearest" type="coq"
      src="left_Binary32_Nearest.v"/>
  </annotation-xml>
</semantics>
```

This shortens somewhat the XML document, as Coq proofs are very lengthy. In addition, if all proofs are saved in a single directory, it is easy to check them all by sending them in a row to the proof checker.

6 Conclusion and Perspectives

In this paper, we have presented the LEMA language, our extension of MathML that suits our needs for code generation and formal proofs of numerical codes. We have exemplified the extension to floating-point arithmetic: representation of floating-point numbers and creation of the `lema:type` attribute, incorporation of their evaluation in floating-point arithmetic through annotations, expressions of floating-point properties through attributes and links to their proof.

Beyond floating-point evaluation, LEMA should express other arithmetic properties like error bounds and value ranges, special values (infinity, signed zero, NaN), and arithmetics and their associated formats. This is currently
under development. At a higher level, specifications, such as a description of hardware capabilities or a function specification, can be reused. Thus, we intend to set up a database mechanism. Meanwhile, we are elaborating an XML schema to validate LEMA documents.

The LEMA language is developed simultaneously with a library that enables to integrate various tools, as shown in Figure 1. Linking with the tools other than Gappa is work in progress.

References


Recent Developments in \Omega\textsc{mega}'s Proof Search Programming Language

Serge Autexier\textsuperscript{a} Dominik Dietrich\textsuperscript{a}

The development of interactive tactic based theorem provers started with the LCF system \cite{Milner_1978}, a system to support automated reasoning in Dana Scott’s “Logic for Computable Functions”. The main idea was to base the prover on a small trusted kernel, while also allowing for ordinary user extensions without compromising soundness. For that purpose Milner designed the functional programming language ML and embedded LCF into ML. ML allowed to represent subgoaling strategies by functions, called tactics, and to combine them by higher order functions, called tacticals. By declaring an abstract type \texttt{theorem} with only simple inference rules type checking guaranteed that tactics decompose to primitive inference rules.

While allowing for efficient execution of recorded proofs by representing them as a sequence of tactic applications, it has been recognized that these kind of proofs are difficult to understand for a human. This is because the intermediate states of a proof become only visible when considering the changes caused by the stepwise execution of the tactics. Tactic proofs can be extremely fragile, or reliant on a lot of hidden, assumed details, and are therefore difficult to maintain and modify (see for example \cite{Boutel_2008} or \cite{Coquand_2009} for a general discussion). As the only information during the processing of a proof is the current proof state and the next tactic to be executed, a procedural prover has to stop checking at the first error it encounters.

\textsuperscript{a}German Research Centre for Artificial Intelligence Bremen (DFKI), \{serge.autexier|dominik.dietrich\}@dfki.de

\textbf{Recent Developments in \Omega\textsc{mega}'s Proof Search Programming Language}

\cite{LeFevre_2000}

\cite{MathML}

\cite{Melquiond_2000}

\cite{Melquiond_2006}

\cite{Moller_1965}

\cite{Muller_2010}

\cite{NetworkWorkingGroup_2001}

\cite{OpenMath2}

\cite{Puschel_2005}

\cite{Puschel_2011}

\cite{Putot_2004}

\cite{Revy_2009}

\cite{Autexier_2010}

\cite{Autexier_2011}

\cite{Bernstein_2012}
This has led to declarative proof languages, inspired by MIZAR [16], where proof steps state what is proved at each step, as opposed to a list of interactions required to derive it. It has been argued that structured proofs in a declarative proof language are easier to read and to maintain. Moreover, as a declarative proof contains explicit statements for all reasoning steps it can recover from errors and continue checking proofs after the first error. It has been noted in [17] that a proof language can be implemented rather independently of the underlying logic and thus provides an additional abstraction layer. Due to its advantage many interactive theorem provers nowadays support declarative proofs (see for example [15, 17, 5, 8]).

The paradigm of proof planning [7], which initially consisted of wrapping LCF-tactics into planning operators with pre- and post-conditions can be viewed as an attempt to exploit the macro-structure of proofs for the specification and organization of proof search. Taking up this idea to integrate domain knowledge to heuristically guide proof search has been at the heart of the research activities of the ΩMEGA group since the beginning. It resulted in concepts to encode common patterns of reasoning, so-called methods, control rules, and strategies, as well as the integration of external specialist systems such as computer algebra systems, classical first order reasoner, or constraint solver. However, the formalisms to design methods, control rules and strategies where objects in ΩMEGA’s underlying programming language, i.e. LISP.

In this paper we want to present the recent developments towards providing an intermediate language as a declarative, abstract layer to specify proof search knowledge, which is independent of the underlying programming language, and motivated by the following reasons: (i) A restricted language is easier to learn and therefore supports the user in writing his own tactics. (ii) Providing rich patterns allows the specification of complex tactics in a compact way. (iii) A separate language can easily be extended and allows the change of internals without breaking the functionality of proof search procedures. Moreover, it is a first step towards exchanging reasoning procedures between different proof assistants.

The starting point to have a well-defined separation line was to install assertion-level reasoning as the basic reasoning layer of ΩMEGA, which provided a suitable abstraction over the base logic. That layer consists of inference rules, which can either be derived from axioms or lemmas (collectively called assertions, see [2] and [3]) and thus are trusted, or else user-defined rules wrapping, for instance, calls to external reasoners, and thus untrusted. The first level of intermediate language concerns the declarative specification of inference rules as well as restrictions on how they are applied during proof search. This will be presented in Section 1. The next step was to provide a declarative language different from the underlying programming language to specify strategies (still more or less in the LCF-tactic style), which provide means to query inference rules from the context and provide search control constructs, and are briefly presented in Section 2. The last step so far was to provide a declarative proof language inspired by ISAR and especially to provide a declarative tactic language to specify strategies that generate declarative proof script, coming closer again to the ideas of operationalizing the proof structuring during proof. The tactic language is presented in Section 3.

A common characteristic of all three levels is to rely on compilation techniques, which result in more efficient code compared to their interpreted counterparts. Furthermore, optimizations can be performed independently of the language itself which increases robustness, and also allows for more efficiency when using domain and problem specific reasoning procedures. We strongly believe that declarative tactic languages offer similar advantages as declarative proof languages, namely robustness and readability, and that the trend towards declarative proof languages will carry on with declarative tactic languages.

1 Annotated Inferences

Inference annotations allow the control of the search space explored in order to enable the application of inference rules, which are ΩMEGA’s basic proof construction steps and which have the distinctive feature that they can be applied deeply, that is on proper subformulas of the formulas occurring in sequents. This allows for exponentially shorter proofs than in classical sequent calculus while increasing the number of possible applicable inference rules at each step. In practical applications, there is therefore a trade-off between the advantage of shorter proofs and the disadvantage that these short proofs may be harder to find, because there are more choice points in the search space. The inference annotation language allows the annotation of inference rules with search space restrictions. Annotated inferences are subsequently compiled into a primitive internal language. The benefits are twofold: (i)
The annotations enable a high-level and fine-grained though declarative control of the internal search procedure, supporting the full range from top level inference to full deep inference as well as a mixture of them. (ii) Compilation results in reasonably efficient proof search procedures comparable to built-in procedures. Let us elaborate on these concepts more precisely.

Intuitively, an inference is a proof step with multiple premises and conclusions augmented by (1) a possibly empty set of hypotheses for each premise, (2) a set of application conditions that must be fulfilled upon inference application, (3) a set of completion functions that compute the values of premises and conclusions from values of other premises and conclusions, such as new meta-variables. Inferences are applied to tasks, which are a multi-conclusion sequents, expressing a subgoal to be proved. Initially there is only a single task consisting of the conjecture to be proved. A proof attempt is represented by an agenda, consisting of a set of tasks, a global substitution which instantiates meta-variables, and contextual information. Tasks are reduced to subtasks. In the case that the subtasks are empty, it is called closed, otherwise it is called open.

A preeminent feature of inferences is that they can be applied deeply inside formulas. Without going into details, let us note that polarities and uniform notation are sufficient to define a uniform notion of a logical context and to dynamically derive so-called replacement rules from that context (see [1] for details).

**Example 1.1.** Consider the application of the axiom rule

\[
\frac{P \quad \text{AXIOM}}{P}
\]

and the following proof situation \( \Gamma, (P^+ \Rightarrow \beta Q^-)^- \vdash Q^+, \Delta \)

Matching the conclusion of AXIOM to the goal \( Q^+ \), and the premise against the \( Q^- \) within \( (P^+ \Rightarrow \beta Q^-)^- \) represents an inference match for the close direction of AXIOM, as both \( Q \) are \( \alpha \)-related via \( \vdash \). This allows us to close the goal \( Q^+ \). However, the use of \( Q^- \) results in a new proof obligation, namely the goal \( P^+ \), as it is \( \beta \)-related to \( Q^- \). Consequently, we get the following transformation:

\[
\Gamma, (P^+ \Rightarrow \beta Q^-)^- \vdash Q^+, \Delta \xrightarrow{\text{AXIOM}} \Gamma, (P^+ \Rightarrow \beta Q^-)^- \vdash P^+, \Delta
\]

In order to enable the application of an inference, candidates for premises respectively the conclusion are searched in a sequent; by default, for a premise or conclusion we search inside all formulas of the sequent for subformulas that unify with the given formula. To control that search annotations within curly brackets can be attached to each premise and conclusion, respectively. Two variants of the axiom rule are shown below:

\[
\frac{P \quad \text{axiom1}}{P} \quad \frac{P[\{\vdash\}] \quad \text{axiom2}}{P}
\]

For **axiom1** the matching candidates of the premise are all top-level formulas on the left-hand side of the sequent, which is the default, whereas **axiom2** searches all subformulas on the left-hand side of the sequent, indicated by the square brackets followed by the pattern restricting the position. Thus only **axiom2** allows for the derivation

\[
\Gamma, F \Rightarrow G \vdash F, \Delta \xrightarrow{\text{axiom2}} \Gamma, F \Rightarrow G + G, \Delta
\]

We use the annotations \( * \) and \( - \) for a premise or conclusion to indicate that a partner must be found for it; in case of \( * \) the matching formula is kept upon inference application while \( - \) requires the matched formula to be replaced. For instance, the alternatively annotated inferences

\[
\frac{[F]}{G \quad \text{impi1}} \quad \frac{[F]}{G \quad \text{impi2}}
\]

both require to match the conclusion \( F \Rightarrow G \), but only the second also requires to remove it upon application. As an effect they result respectively in the following derivations:

\[
\Gamma, F \vdash G, F \Rightarrow G, \Delta \xrightarrow{\text{impi1}} \Gamma \vdash F \Rightarrow G, \Delta
\]

and

\[
\Gamma \vdash G, \Delta \xrightarrow{\text{impi2}} \Gamma \vdash F \Rightarrow G, \Delta
\]
In order to identify a partner formula, higher-order unification is used in general. To restrict this for instance to HO-matching but also to more specific algorithms, such as for the application of inference rules, the keyword check allows to indicate a specific algorithm. Additional restrictions on positions can be expressed either by meta-predicates, or by more specific patterns on sequents.

The next class of annotations is to specify how the formula is traversed when looking for subformulas. By default the list of all candidate subformulas is returned. Alternatively one can use the annotations outermost or innermost in combination with either leftmost or rightmost that the search should stop on the first match: this can for instance be used to specify inferences to be used for a simplification using a leftmost, innermost strategy. Thus, the inference

\[ \frac{P}{P} \text{[leftmost, outermost]} \]

prefers the formula \( F_1 \) over the formula \( F_2 \) in the proof situation \( P \lor F_1, Q \land F_2 \vdash F_3 \). However, the first occurrence of \( F \) introduces a proof obligation \( \neg P \), while the second occurrence of \( F \) can be used without the introduction of new proof obligations. To restrict the applicability of the axiom rule to the latter, we provide the annotation nopob.

2 Declarative Language for Procedural Strategies

On top of inferences, proof assistance systems provide the layer of tactics to organize the proof search. These tactics are typically written in the underlying programming language and require the user to know about the internal data structures and functions. This not only hinders people not knowing these details from writing their own search procedures, but also makes third party tactic libraries not maintained by the proof assistance developers fragile with respect to changes of the underlying system in the sense that they may not compile anymore. To bridge the gap between the predefined proof operators and the programming language of the proof assistant we developed the tactic language CRStL [9] as an intermediate abstraction layer. This new layer of abstraction can be seen in analogy to what has been done by introducing declarative proof languages. It is clear that for complex and efficient proof strategies an expressive programming language is necessary: for that reason CRStL is extensible by user defined functions and predicates written in the underlying language of the prover. However, small parts of proofs can often be automated by special purpose proof strategies, where a much simpler language suffices. It is desirable that a user can write such a strategy on the fly and requiring the use of the underlying programming language in such a case often prevents the non-expert user to take this option because he is unfamiliar with programming in the systems language. Even for experienced users it is often too time consuming to design a special purpose proof strategy in the underlying programming language when its use will be restricted only to a small part of a theory.

The language is arranged in two levels, a query language to access mathematical knowledge maintained in development graphs [12], and a strategy language to annotate the results of these queries with further control information. This comes from the insight that restricting the number of proof operators drastically reduces the search space and allows for a more efficient solution computation, provided that the filter is not too strict (see [14] and [13] for related work on relevance filtering). Therefore, the language explicitly supports the cycle select - process - search. Note that by the introduction of queries/filters tactics become dynamic objects that depend on the context. We call such adaptive tactics theory-aware. An example is shown in Figure 1, which shows a simplification strategy that selects all orientable equations from the current theory and applies them as long as possible.

The control layer emphasizes on providing language constructs to specify and integrate conditions in form of declarative patterns and meta-predicates on sequents, including the handling of subformulas and polarities. These conditions are used to specify applicability conditions of tactics, case analysis, or termination conditions for loops. Additionally, backtrack conditions can separately be installed to avoid the exploration of certain branches of the search space. An example is shown in Figure 2, which shows a simple tactic that performs a forward exploration to derive a subformula given as parameter of the tactic. Finally, Figure 3 gives an example in which the selected knowledge is further post processed using a Knuth-Bendix completion procedure.

3 Declarative Language for Declarative Proof Script Strategies

The strategies from the previous section are still in the realm of LCF-tactics, which take a set of goals (agendas) and return a new, possibly empty set of goals. In [4] we presented a declarative tactic language on top of a declarative proof
strategy simplification
    repeat
    use (select lhs=rhs from current
        where (greaterlpo lhs rhs)) as forward union
    use (select lhs=rhs from current
        where (greaterlpo rhs lhs)) as backward

Figure 1: Selecting and annotating knowledge from current theory

strategy fwexplore
    parameter formula
    repeat
    use select * from current.inferences as forward
    until *,[formula]- |- *
        where (not (proofobligations (pos formula)))
    backtrack-if (greater stratdepth 3)

Figure 2: Forward exploration with declarative termination condition

language (which can be seen as an extension of [9]). Our language comes along with a rich facility to declaratively specify proof states (and conditions on them) in the form of sequent patterns, as well as ellipses (dot notation) to provide a limited form of iteration. The language differs from the language for procedural strategies in that the strategies are defined using an extended declarative proof script language by specifying intermediate proof states of the proof construction. These intermediate proof states act as islands or stepping stones between the assumptions and the conclusion (by omitting the constraints indicating how to find a justification of the proof step) leaving the task of closing the gaps to automation tools. The execution of a declarative tactic results in a declarative proof script, which in turn can be inserted into the document if desired. Thus, they provide a means to overcome the main problem of the declarative style of proof, namely that it is laborious to write and thus close the gap between both proof styles. Moreover, because of its additional abstraction, it might provide possibilities to exchange reasoning procedures between different proof assistants in the long-term view. Indeed, it has been noted in [17] that a declarative proof language can be implemented rather independently of the underlying logic.

As an example, we consider the problem of proving the theorem \( \lim_{x \to 3} (x^2 - 5x - 2) = 4. \) After expanding the definition of \( \lim \), the proof state consists of the two goals \( \epsilon > 0, |x - 3| < |\delta| \) and \( \epsilon > 0 \) \( \vdash |x^2 - 5x - 2 - 4| < \epsilon \). The declarative proof script is shown at the top of Figure 5, where the declarative tactic factorbound (see Figure 4) is not yet processed. Processing the factorbound-statement expands it and results in the following steps:

1. The pattern of the cases condition is matched, yielding the following binding: \( \{ \text{LHS} \mapsto x - 3, \text{RHS} \mapsto \delta, \text{GOALLHS} \mapsto \frac{x^2 - 5}{x - 2} - 4, \text{GOALRHS} \mapsto \epsilon \} \)
2. To be able to evaluate the where condition, the first with part is evaluated. This results in the following factorization: \( Y_1 \times \ldots \times Y_n = (x - 3) \times (\frac{x - 5}{x - 2}) \times (x - 1) \). Internally, a list \( Y = [(x - 3), (\frac{x - 5}{x - 2}), (x - 1)] \) is generated and \( n \) is bound to 3. In the next assignment \( j \) is bound to 1 by looking up \( x - 3 \) in the list of factors.
3. The conditions of the where part evaluate to true.
4. The with part of the proof is evaluated, generating a list \( M = [\delta, MV1, MV2] \) of length 3.
5. The proof part is expanded and inserted, resulting in the proof script shown at the bottom in Figure 5.

The example illustrates the following points: The declarative strategy is easy to specify as the overall problem can easily be divided in a sequence of subproblems. Due to the use of patterns, the strategy is independent of the number of factors computed by the computer algebra system. More importantly, it allows the formulation of what to do next after the factorization, namely to bound all but one factor.
strategy KBGroup
use complete select lhs=rhs from groups as forward

Figure 3: Post-processing selected knowledge using Knuth Bendix completion

strategy factorbound
cases
  abs(LHS)<RHS,* |- abs(GOALLHS) < GOALRHS
  where (and (variable-eigenvar.is "GOALRHS")
    (metavar-is "RHS")
    (some #'(lambda (x) (term= "LHS" "x"))
      "Y\_j" .. "Y\_N")
  with Y\_1 * .. * Y\_N = (maxima-factor "GOALLHS")
  j = (termposition "LHS" "Y\_1 .. Y\_N")
->
proof
L1: GOALLHS= Y\_1 * .. * Y\_N by abelian decide
    foreach i in 1..N where (not (= "j" "i"))
      Y\_j <= MV\_j by linear bound
  end
L2: abs(GOALLHS)=abs( Y\_1 * .. * Y\_N) from L1
    .<= abs(Y\_1) * .. * abs(Y\_N)
    .<= MV\_1 * .. * MV\_N
    .<= GOALRHS
qed
with foreach i in 1..N
  M\_i = (if (= "i" "j") "RHS"
    (make-metavar (term-type "RHS")))

Figure 4: Dynamic pattern matching and proof script generation

4 Summary and Outlook

We have briefly reviewed the different concepts and language levels developed during our overall quest towards an declarative, abstract layer to specify proof search knowledge, which is independent of the underlying programming language of a proof assistant system. These are implemented in an experimental version of the ΩMEGA proof assistance system. Ongoing research is devoted to extend these ideas to the specification language layer and allow for the specification of tactics that generate or transform specifications, but also tactics that transform both specifications and proof scripts in combination.

References

Vol. 44, No. 2, June 2010

Recent Developments in \(\Omega\)mega’s Proof Search Programming Language

\[ \text{theorem th1: } \lim_{x \to 3} \frac{x^2 - 5}{x - 2} = 4 \]

\[ \text{proof} \]

\[ \text{subgoals} \]

\[ \text{subgoal } |\frac{x^2 - 5}{x - 2} - 4| < \varepsilon \text{ using A1: } \varepsilon > 0 \text{ and A2: } |x - 3| < ?\delta \]

\[ \text{by factorbound} \]

\[ \text{subgoal } ?\delta > 0 \text{ using } \varepsilon > 0 \]

\[ \text{end by limdefbw} \]

\[ \text{qed} \]

\[ \text{theorem th1: } \lim_{x \to 3} \frac{x^2 - 5}{x - 2} = 4 \]

\[ \text{proof} \]

\[ \text{subgoals} \]

\[ \text{subgoal } |\frac{x^2 - 5}{x - 2} - 4| < \varepsilon \text{ using A1: } \varepsilon > 0 \text{ and A2: } |x - 3| < ?\delta \]

\[ \text{proof} \]

\[ L1: \frac{x^2 - 5}{x - 2} - 4 = (x - 3) \cdot \left(\frac{1}{x - 2}\right) \cdot (x - 1) \text{ by abeliandecide} \]

\[ |x - 1| \leq ?MV1 \text{ by linearbound} \]

\[ \left|\frac{1}{x - 2}\right| \leq ?MV2 \text{ by linearbound} \]

\[ L2: \left|\frac{x^2 - 5}{x - 2} - 4\right| \leq |(x - 3) \cdot \left(\frac{1}{x - 2}\right) \cdot (x - 1)| \text{ from L1} \]

\[ \leq |x - 3| \cdot \left|\frac{1}{x - 2}\right| \cdot |x - 1| \]

\[ \leq ?\delta \cdot ?MV1 \cdot ?MV2 \]

\[ \leq \varepsilon \]

\[ \text{qed} \]

\[ \text{subgoal } ?\delta > 0 \text{ using } \varepsilon > 0 \]

\[ \text{end by limdefbw} \]

\[ \text{qed} \]

Figure 5: Declarative proof script of the example before and after processing the call of the declarative tactic factorbound


58
transalpyne: a language for automatic transposition

Luca De Feoa  Éric Schostb

Abstract

We present here transalpyne, a scripting language, to be executed on top of a computer algebra system, that is specifically conceived for automatic transposition of linear functions. Its type system is able to automatically infer all the possible linear functions realized by a computer program. The key feature of transalpyne is its ability to transform a computer program computing a linear function in another computer program computing the transposed linear function. The time and space complexity of the resulting program are similar to the original ones.

1 Introduction

Computer Algebra is devoted to developing algorithms to work on symbolic representations of mathematical objects. Linear maps over vector spaces or, more generally, free modules are often represented by matrices, either in dense or sparse form. The so-called black-box model gives another way of representing a linear application $L : V \rightarrow W$: a computer program that on any input $v \in V$ gives as output $L(v)$ is taken as a symbolic representation of $L$; this of course assumes a precise computer representation of the elements of $V$ and $W$.

Since the seminal paper [25], computer algebraists have developed algorithms to work with black-box represented linear maps. In the black-box model, algorithms are only allowed to query the black-box by feeding an input to the black-box program and reading its output; no other information on the linear map can be obtained, in particular the source code of the program cannot be analyzed. The complexity of black-box algorithms is measured as in the computational model being used to describe the algorithm, plus the number of calls to the black-box program is taken into account as a special parameter.

In the black-box model, algorithms are known to compute the minimal polynomial, the determinant, the inverse, the rank [25, 17] and the characteristic polynomial [6, 23, 5]. This model is interesting whenever the matrix representing the linear map is too big to allow efficient processing by a computer program, however its information can easily be compressed in a black-box program: sparse or Vandermonde matrices are a classical example.

---

[a]LIX, École Polytechnique, France, luca.defeo@polytechnique.edu
[b]ORCCA, CSD, University of Western Ontario, Canada, eschost@uwo.ca
On the other side, Algebraic Complexity studies the complexity of computer programs that perform algebraic computations by abstracting from the actual representation of algebraic elements. Only arithmetic operations in the ring of interest are accounted for. One of the standard models used in algebraic complexity is the arithmetic circuit: a directed acyclic graph (DAG) is used to represent the flow of arithmetic evaluations, each node of the DAG accounts for one arithmetic operation (usually + or ∗).

In particular, arithmetic circuits can be used to represent black-box programs computing linear maps. Then it is a well known theorem, [2, Th. 13.20] that a linear map and its transpose have similar algebraic complexities in the arithmetic circuit model; this is often known as transposition theorem or Tellegen’s theorem. By a well known equivalence [2, Lemma 13.17], the transposition theorem extends to the straight-line program (SLP) model. These results justify the fact that some extensions of the black-box model allow black-box algorithms to query the linear form as well as its transpose.

Besides that, extensions of the transposition theorem to the Random Access Machine (RAM) model have been successfully applied in computer algebra to develop efficient algorithms [22, 1, 12, 19, 4]. The key to all these results is to realize that a certain map is the transpose of some other well known linear map \( L \). Then, an efficient algorithm in the RAM model for \( L \) is translated to the arithmetic circuit model, the transposition theorem is applied and the resulting arithmetic circuit is translated back to a RAM algorithm. All the papers use ad hoc transformations to/from the arithmetic circuit model but give no general technique to perform such translation; the only notable exception is [1] that defines a very restricted language—not far away from the SLP paradigm—in which a constructive proof of the transposition principle is possible.

Here we present an extension of [1] that allows to automatically treat a wider class of programs. We reserve the theoretical details of the construction for a forthcoming paper and focus instead on its implementation. We are currently developing an extension language for python, called \texttt{transalpyne}, for which transposition can be automatically performed by the compiler/interpreter.

One of the key features of \texttt{transalpyne} is the possibility to automatically discover all the possible linearizations of a program. In fact, many linear functions can correspond to the same computer program: in the case of multiplication of polynomials, for example, the same program corresponds to two linear functions, namely left and right multiplication by a constant. \texttt{transalpyne} uses an algorithm similar to the type inference of statically typed functional languages [3] to discover all of these linearizations.

For each discovered linearization, the compiler/interpreter generates the correct transposition. It can be shown that the algebraic complexity of the resulting program is similar to the one of the original program. In the next sections we summarize the theory and the practice of transposition in \texttt{transalpyne}.

## 2 Arithmetic circuits

In this section we briefly present the arithmetic circuit model. For convenience, our presentation slightly deviates from textbooks; for a more classical and extensive treatment see [2, 24].

### 2.1 Basic definitions

**Definition 1** (Arithmetic operator, arity). Let \( R \) be a ring. An arithmetic operator over \( R \) is a function \( f : R^i \to R^o \) for some \( i, o \in \mathbb{N} \). We set \( R^0 = 0 \), the zero \( R \)-module. Here \( i \) is called the \textit{in-arity} of \( f \) or simply \textit{arity}, \( o \) is called the \textit{out-arity} of \( f \).

**Definition 2** (Arithmetic basis). Let \( R \) be a ring. An arithmetic \( R \)-basis is a (not necessarily finite) set of arithmetic operators over \( R \).

The arithmetic basis we will work with is the \textit{linear basis}, denoted by \( \mathcal{L} \). It is composed of

\[
\begin{align*}
+: R \times R &\to R \quad &*: a : R &\to R \\
a, b &\mapsto a + b \quad &b &\mapsto ab \quad &a &\mapsto a, a \\
0 : 0 &\to R \quad &\omega : R &\to 0 \\
\bot &\mapsto 0 \quad &a &\mapsto \bot 
\end{align*}
\]

(\( \mathcal{L} \))
where we denote by ⊥ the unique element of 0 to avoid confusion with the 0 of \( R \). Arithmetic circuits are directed acyclic multigraphs carrying information from an arithmetic basis; the formal definition follows.

**Definition 3** (Arithmetic node). Let \( R \) be a ring and \( B \) be an \( R \)-basis. A node over \((R, B)\) is a tuple \( v = (I, O, f) \) such that

- \( I \) and \( O \) are finite ordered sets,
- \( f \) is either an element of \( B \) or the special value \( \emptyset \).
- If \( f = \emptyset \), one of the two following conditions must hold:
  - \( I \) is a singleton and \( O \) is empty, in this case we say that \( v \) is an input node;
  - \( I \) is empty and \( O \) is a singleton, in this case we say that \( v \) is an output node.
- If \( f \neq \emptyset \), the cardinality of \( I \) matches the in-arity of \( f \) and the cardinality of \( O \) matches the out-arity of \( f \); in this case we say that \( v \) is an evaluation node.

We call input ports the elements of \( I \) and output ports the elements of \( O \), which we denote respectively by \( \text{in}(v) \) and \( \text{out}(v) \). The cardinalities of \( I \) and \( O \) are called, respectively, the in-degree and out-degree of \( v \). We call \( f \) the value of \( v \) and write \( \beta(v) \) for it.

**Definition 4** (Arithmetic circuit). Let \( R \) be a ring and \( B \) be an \( R \)-basis. An arithmetic circuit over \((R, B)\) is a tuple \( C = (V, E, \preceq, \leq_i, \leq_o) \) such that

1. \( V \) is a finite set of nodes over \((R, B)\);
2. \( \prec \) is a total order on \( V \), \( \leq_i \) is a total order on the input nodes in \( V \), \( \leq_o \) is a total order on the output nodes in \( V \);
3. let \( I = \bigcup_{v \in V} \text{in}(v) \) and \( O = \bigcup_{v \in V} \text{out}(v) \), then \( E \) is a bijection from \( O \) to \( I \) such that \( E(o) = i \) implies that \( o \in \text{out}(v), i \in \text{in}(v') \) and \( \leq_o \leq v' \).

It is useful to see \( E \) as a set of pairs \((o, i)\) with \( i \in I \) and \( o \in O \). Then the elements of \( E \) are called the edges of the circuit. The edges incident to \( v \in V \) are all the \((o, i) \in E \) such that \( i \in \text{in}(v) \); the edges stemming from \( v \in V \) are all the \((i, o) \in E \) such that \( o \in \text{out}(v) \). An edge stemming from \( v \) and incident to \( v' \) is said to connect \( v \) to \( v' \). We call inputs and outputs of a circuit, respectively, the input and output nodes in \( V \); which we denote by \( \text{in}(C) \) and \( \text{out}(C) \).

**Definition 5** (Size, depth). Let \( C \) be a circuit over \((R, B)\). The size of \( C \), denoted by \( \text{size}(C) \), is the number of evaluation nodes in \( V \); the depth of \( C \), denoted by \( \text{depth}(C) \), is the length of the longest directed path in a graph-theoretic sense in \((V, E)\).

Sometimes it is useful to only count certain nodes. Let \( X \subset B \), the \( X \)-weighted size of \( C \), denoted by \( \text{size}_X(C) \), is the number of nodes \( v \in V \) such that \( \beta(v) \in X \).

Figure 1 shows an example of arithmetic circuit, the analogy with multiDAGs is evident. We draw input and output nodes in square boxes and evaluation nodes in round boxes.

Circuits are endowed with the usual semantics consisting in the evaluation of the arithmetic operations on their inputs. We denote by \( \text{eval}_C \) the function \( R^i \to R^o \) computed by the circuit \( C \). For a circuit over \((R, L)\) it can be shown that \( \text{eval}_C \) is a linear operator; we skip the formal definitions and proofs for conciseness.

### 2.2 The transposition theorem

For a circuit over the basis \( L \), each node can be viewed as a linear operator and the arrows can be understood as composing operators in a suitable way to obtain \( \text{eval}_C \). By reversing the flow and transposing the operator computed at each node, one obtains a circuit that computes the transposed operator.
Definition 6. Dual circuit Let $C = (V, E, \leq, \leq_I, \leq_O)$ be a circuit over $(R, \mathcal{L})$, the dual circuit of $C$, denoted by $C^*$, is the arithmetic circuit

$$C^* = (V^*, E^{-1}, \leq', \leq'_I, \leq'_O)$$

where for any node $v = (I, O, f)$ in $V$ there is a node $v^* = (O, I, f^*)$ in $V^*$ where

$$f^* = \begin{cases} f & \text{if } f = *_a, \\ + & \text{if } f = H, \\ H & \text{if } f = +, \\ \omega & \text{if } f = 0, \\ 0 & \text{if } f = \omega; \end{cases}$$

and for any input node $v = (\emptyset, O, \emptyset)$ there is an output node $v^* = (O, \emptyset, \emptyset)$ and vice versa. The orderings $\leq', \leq'_I$ and $\leq'_O$ are defined as follows:

$$v \leq v' \iff v'' \leq' v^*,$$

$$v \leq'_I v' \iff v^* \leq'_O v'^*,$$

$$v \leq'_O v' \iff v^* \leq'_O v'^*.$$

In particular, this makes $(V', E^{-1})$ the reverse graph of $(V, E)$ in a graph-theoretic sense. Figure 1 shows two circuits that each other’s dual. We now state the transposition theorem, for a proof see [2, Th. 13.20].

Theorem 1 (Transposition theorem). Let $C$ be a circuit over $(R, \mathcal{L})$ that computes a linear application $f$, then $C^*$ computes the transposed linear application $f^*$.

Corollary 1. A linear function $f : R^n \rightarrow R^m$ and its transpose can be computed by arithmetic circuits of same sizes and depths. In particular if $C$ computes $f$ and $C^*$ computes $f^*$,

$$\text{size}_{(+)}(C) = \text{size}_{(H)}(C^*), \quad \text{size}_{(H)}(C) = \text{size}_{(+)}(C^*),$$

$$\text{size}_{(+\omega)}(C) = \text{size}_{(\omega)}(C^*), \quad \text{size}_{(\omega)}(C) = \text{size}_{(+)}(C^*).$$

A circuit is limited to compute one specific function with inputs and outputs of fixed size (in term of elements of $R$). However complexity theory is interested in algorithms that compute on inputs of variable size. This leads to study families of circuits.

Definition 7 (Circuit family). Let $R$ be a ring, $\mathcal{B}$ a basis over $R$ and $\mathcal{P}$ a set. A circuit family over $(R, \mathcal{B}, \mathcal{P})$ is a family of circuits over $(R, \mathcal{B})$ indexed by $\mathcal{P}$. $\mathcal{P}$ is called the parameter space of the family. When the mapping from $\mathcal{P}$ to the circuits is Turing-computable, the family is called uniform.

We are mainly interested in uniform circuit families since they are equivalent to computable functions, theorem 1 easily generalizes to them. We will not study uniform circuit families more in depth, what we do instead is directly work on computer programs implicitly representing circuit families and automatically deduce the transposed family without actually using the circuit model. More details on uniform circuit families can be found in [24].
3 transalpyne

transalpyne is a programming language suitable for expressing linear algebraic programs and automatically transpose them. Its compiler/interpreter is able to implicitly deduce which families of circuits a given program is equivalent to and to produce a program computing the transposed family.

3.1 Concepts

transalpyne has been conceived as a scripting language to be used on top of computer algebra systems. We made an effort to give syntax and semantics as close as possible to the python programming language.

In transalpyne there is no such concept as an executable program: only functions can be defined in transalpyne. We call target language the language to which transalpyne programs are compiled; the output of compilation is a library file whose functions can be imported by programs written in the target language. Only compilation to python is supported for the moment.

transalpyne can also be interpreted via the python interpreter. A transalpyne library contained in a file my-library.yp can be imported in a python program via the statement

```python
import my-library
```

The python interpreter recognizes the .yp extension and launches the transalpyne interpreter on the file; the functions of the library are interpreted and transposed by the transalpyne interpreter and their names are exported to the python namespace.

transalpyne is mostly dynamically typed, with the only exception of algebraic types. In order to transpose a function, transalpyne must know at transpose time which variables contain algebraic elements and which variables contain other data (such as booleans, strings, ints, etc.); this can be done by explicitly specifying the type of the input and output parameters of a function, while all the other variables can be left untyped. transalpyne supports two sorts of algebraic types: ring elements and module elements; we plan to support more complex algebraic types, such as algebras, in the future. transalpyne relies on python’s operator overloading to represent ring operations.

Before transposing a function, transalpyne must prove that it is indeed a linear function in its arguments. The technique it uses is to linearize the function, that is to make certain input and output parameters constant until it can be shown that the remaining output parameters are linear in the remaining input parameters. We discuss this in Section 4.

3.2 Syntax

We only describe transalpyne syntax informally. Indentation has a syntactic value (it delimits blocks) and keywords are pretty much the same. A transalpyne file contains a type declaration section followed by a name definition section.

3.2.1 Type declarations

transalpyne supports two type constructors: a ring constructor and a free module constructor.

```transalpyne
type Ring R
type Module(R) M
```

This example declares R as a ring type and M as a free module type over the ring R. The typechecker ensures that modules are consistently declared.
3.2.2 Name declarations

Name declarations take three forms: imports, function definitions and aliases. Imports are declared as in python and have the same semantics. Note however that the linearization algorithm considers any imported function as a constant function.

There is no return statement in transalpyne, function definitions are declared as follows

```python
def (a, b) my_function(c, d):
```

where input arguments are given on the right and output arguments on the left.

Inside function definitions, there are four types of statements: pass statements\(^1\), assignments (including augmented assignments), for loops and if s. The syntax is identical to python’s.

On the left hand side of assignments, may only appear variable names and subscripts. On the right hand side of assignments, the following types of expressions may appear:

- String, numeric and boolean constants;
- Binary and unary operators +, -, *, /, %, div, mod, <, >, <=, ==, !=, and, or, not, in;
- Parenthesized expressions;
- Subscripts and slices;
- List constructors, including comprehensions;
- Variable evaluations;
- Function calls.

The syntax for all of these is identical to python’s. The only notable exception are function calls where a keyword trans is added to let the user call a transposition of a function. In case a function has more than one linearization (and thus more than one transposition), signature specifiers enclosed in braces \{, \} permit to specify which linearization/transposition is wanted.

Finally, aliases permit to export specific linearization/transpositions of functions with names that can be used inside a python program.

Figure 2 gives a complete transalpyne example. It defines a product function and two aliases (with transposition and signature specifiers).

```python
type Ring R
def (R c) product(R a, R b):
    c = a * b
l_product = trans \{linear R\} product{linear R, const R}
r_product = trans \{linear R\} product{const R, linear R}
```

Figure 2: A transalpyne program

3.3 Semantics

We only give here the points were transalpyne semantics differ from python’s.

\(^1\)The statement that does nothing.
### 3.3.1 Types

Transalpyne is statically typed for algebraic types. The type of each input and output parameter of a function must be specified in the definition as in figure 2. When the type of an argument is omitted, it is assumed to have non-algebraic type. Variables inside the body of a function cannot be explicitly typed, a type-inference algorithm deduces their types from the types of the input parameters.

### 3.3.2 Side effects

There is no side effect in transalpyne. In particular, there is no global variable and assignment itself is more akin to the let-binding of a functional language. After having transposed the functions, the transalpyne compiler/interpreter leaves to the target language the task of executing them, thus it cannot enforce the no-side-effect policy at runtime. It is the responsibility of the user to insure that no side effect happens inside a transalpyne function.

### 3.3.3 Algebraic variables

Type declarations merely say that some variables belong to a type, but do not specify any particular implementation of the type. The implementation of rings and modules is left to the user and must be given in an external module written in the target language. The user is only required to implement them as objects and to expose a few methods.

Ring objects must:

- Overload + and * with the obvious semantic;
- Implement a method zero that returns the zero of the ring;
- Optionally, implement a method one that returns the one of the ring;
- Optionally, implement a method \( \mathbb{Z} \) that takes an integer \( n \) and returns the element \( n \cdot 1 \) of the ring;
- Optionally, implement methods div and mod that perform Euclidean division with remainder;
- Optionally, overload / thus making the ring into a field.

Module objects must:

- Overload + and * with the obvious semantic;
- Implement a method zero that returns the zero of the module;
- Overload the subscript operator \([\cdot]\) so that it implements some arbitrary projections on the underlying ring. Most often, a module will be implemented as an array of ring objects and \([i]\) will just be projection onto the \(i\)-th coordinate.
- Overload the assignment to subscript operator in the obvious way.

Algebraic output parameters of a function are implicitly initialized to zero via their zero method. This insures that non-assigned algebraic output parameters are always linear in the inputs of the function.

Algebraic elements cannot be combined through the use of lists: lists of algebraic objects are non-algebraic objects and extraction from a list always yields a non-algebraic object.

### 3.3.4 Function calls

Transalpyne does not have tuples; the return type of a function with many output parameters is not a tuple, as a consequence its return value cannot be assigned to a variable: it must be assigned to as many variables as there are output parameters. Another consequence of this is that functions with many outputs cannot be used inside expressions: their outputs can only be assigned to variables.

Function names not declared in the library are simply regarded as external functions. They are assumed to have one return parameter, thus a multi-assignment will return an error. External functions have no algebraic input or output parameters. This is useful to call builtin python functions\(^2\) from inside a transalpyne program.

\(^2\)One common example is the function range, needed to iterate over module elements.
3.3.5 Recursion and Higher order

transalpyne allows recursion and even calling its own transpose. It does not allow to pass functions as arguments to a function, although the transposition algorithm internally uses this technique to transpose for loops. A higher order transposable language is theoretically possible and we plan to add this feature to transalpyne in the future.

4 Linearization

The function

```python
def (R c) product(R a, R b):
c = a * b
```

is not linear in a and b, but it can be made linear by fixing one of the two arguments, for example by considering it as the family of mappings $a \mapsto ab$ for any given b. We call const the arguments that are fixed and linear the others; clearly const outputs must only depend on const inputs, while linear outputs must linearly depend on linear inputs for any given values of the const inputs. This is equivalent to model the function as a family of circuits whose parameter space is the domain of the const arguments.

transalpyne allows the user to annotate the types of the arguments in order to specify whether they are const or linear (non-algebraic arguments are by default const).

```python
def (linear R c) product(linear R a, const R b):
c = a * b
```

Fortunately, the user need not specify all the modifiers since they can be inferred algorithmically.

We call signature a list of linear/const modifiers attached to the arguments of a function. A function can, of course, have more than one signature. The idea behind the signature inference algorithm is simple. transalpyne starts from a few axioms on the signature of elementary operators, here is some of them:

- $\times : (\text{linear}, \text{const}) \rightarrow \text{linear}$
- $\times : (\text{const}, \text{linear}) \rightarrow \text{linear}$
- $(\text{const}, \text{const}) \rightarrow \text{const}$
- $+ : (\text{linear}, \text{linear}) \rightarrow \text{linear}$
- $(\text{const}, \text{const}) \rightarrow \text{const}$
- zero : linear
- const
- one : const

Then transalpyne applies an algorithm similar to the type inference of functional languages [3] to deduce all the possible signatures of a function. If more than one linearization exists, transalpyne will generate one transposition for each of them. The user is also allowed to only specify partial information, the compiler/interpreter will restrict to the signatures that match such information or issue an error if no signature matches.

Function calls and aliases use the same principle. The signature specifiers {...} let the user specify which of the linearizations of a given function has to be called or saved under a new name. Thus, in the example we gave in figure 2, \_product is an alias for the transposed left-linear product, while \_product is an alias for the transposed right-linear one. Aliases are extremely useful since they permit to export to the namespace of the target language the transposed functions that could not be accessed otherwise.
5 A word about automatic differentiation

Before discussing the way transalpyne transposes programs, we recall some concepts from the theory of Automatic Differentiation (AD).

The transposition principle has often been viewed as a special case of the reverse mode in automatic differentiation [18, 14, 1]. This is somewhat ironic as the whole idea of automatic differentiation can elegantly be derived in the arithmetic circuit model and reverse mode in particular is just an application of the transposition principle [7]. It is probable that the need for efficient AD tools in many scientific areas other than mathematics and computer science is responsible for such reversal of roles.

Here we show how AD can be expressed in the arithmetic circuit model and then discuss the main differences between the AD tools and our approach. A much more complete study on the differentiation of circuits and on how the transposition principle relates the gradient to the differential can be found in [7, 20].

To simplify, we consider a basis $\mathcal{B}$ over $\mathbb{R}$ made exclusively of everywhere continuously derivable functions (w.r.t the standard metric of the Euclidean space $\mathbb{R}^n$). What we give here is a technique to approximate a circuit over $(\mathbb{R}, \mathcal{B})$ by a “linear” circuit.

**Definition 1** (Derivative of a circuit). Let $C$ be a circuit over $(\mathbb{R}, \mathcal{B})$ with $n$ inputs and let $x \in \mathbb{R}^n$. For any function $f \in \mathcal{B}$, we denote by $J_f$ its Jacobian. Then the derivative of $C$ at $x$, denoted by $d_x C$ is the arithmetic circuit where any $v \in V$ with $\beta(v) = f$ and incident edges $e_1, \ldots, e_m$ has been substituted by a $v'$ with

\[
\beta(v') = J_f(\mathrm{eval}_{e_1}(x), \ldots, \mathrm{eval}_{e_m}(x)).
\]

![Figure 3: A circuit and its derivative at the point $x = (a, b, c)$.](image)

Taking the derivative of a circuit at $x$ amounts to chose for each node a linear approximation at the point where it is evaluated. It is clear that this yields a linear approximation for the circuit at $x$.

**Proposition 1.** $\mathrm{eval}_{d_x C} = J_{\mathrm{eval}_C}(x)$.

It is also clear that $d_x C$ is defined over a basis that is exclusively made of matrices with coefficients in $\mathbb{R}$. These circuits are slightly more general than those over the basis $\mathcal{L}$, but it is easy to generalize theorem 1 to them. In other words we have defined a transformation from circuits computing derivable functions to linear circuits.

Now $d_x C$ can be queried by black-box algorithms to obtain information about the Jacobian $J_{\mathrm{eval}_C}(x)$. The simplest application is to compute the directional derivative in $x$ along a direction $u$: for this task it suffices to evaluate the circuit once, since $\mathrm{eval}_{d_x C}(u)$ is the desired value. Computing the derivative along $n$ linearly independent directions yields the whole Jacobian and this roughly corresponds to the direct mode in automatic differentiation\(^3\).

When the circuit has many inputs but only one output, there is a more convenient way to get the whole gradient with only one black-box query: $d_x C$ computes a linear form whose coefficients are exactly the coefficients of the gradient.

\(^3\)To be more precise, direct mode automatic differentiation constructs $d_x C$ and evaluates the $n$ directions in parallel, thus reducing the amount of storage needed.
thus the dual circuit \((d_x C)^*\) computes the transposed form, or column vector. The single query \(\text{eval}_{(d_x C)^*}(1)\) yields this vector. This is exactly what is called “reverse mode” in automatic differentiation.

Note however that one is not limited to direct or reverse mode: any black-box algorithm can be combined with the derivative circuit to obtain information on the original function. For example Wiedemann’s algorithm [25] can be used to determine if the function is invertible around \(x\), and the directional derivatives of the inverse can be computed.

Of course, direct and reverse automatic differentiation can be defined by the more classical chain rule, and then the transposition theorem can be derived as a special case of the reverse mode by observing that, when all the nodes of the circuit are linear maps, \(C = d_x C\) for any \(x\). After all, the code transformation techniques given in [1] and developed in the next section were already invented by researchers in AD [10], though not often implemented.

So, why invent transalpyne when there is already plenty of AD tools out there? The answer is manifold and we only list here some key points.

- AD is often interested in recovering the full Jacobian, instead of just having a black-box for it. For an \(n \times m\) matrix, this requires \(n\) queries in direct mode or \(m\) queries in reverse mode. In both cases, AD tools do more work than what we would like to.
- Many AD tools do not optimize the computation of \(d_x C\) for the case where nodes are linear and still compute the whole circuit. In particular, many AD tools generate a graph representation of an arithmetic circuit from a program instead of directly transposing the code. This adds a constant overhead to the case of transposition where simply \(d_x C = C\).
- If the circuit \(d_x C\) is computed, it must be fully stored in memory for reverse mode. This may seem innocuous as \(d_x C\) has the same size as \(C\), but consider programs that compute \(\text{eval}_C\) by means of for loops or other iterative constructs: while the evaluation of \(C\) is compact and cheap, the evaluation of \(d_x C\) possibly requires to introduce a new variable for each iteration of the loop. Depending on the implementation, this may lead to code or storage bloat. In the case of transposition, this never happens since for loops are directly reversed (at least when all the variables are linear). Griewank [11] gives a time/memory compromise that permits to keep both storage and time in a factor of \(\log n\) from the original program, but this is still not as good as transposition.
- Our approach is more general in that it permits to automatically treat functions that depend both on linear and non-linear arguments without any help from the user. Thanks to this, we are able to treat recursive functions, while only few AD tools can.
- Our approach is algebraic and permits to prove bounds on the algebraic complexity of the generated programs, while AD tools usually only deal with floating point numbers. More generally, AD languages are usually less rich than transalpyne.

6 Transposition

After the linearization phase, transalpyne generates the transposed functions. Linear in- and out- arguments are swapped, while const arguments do not move. The body itself of the function is transformed: formally, it is translated to a family of arithmetic circuits, the circuits are reversed and the result is translated back to a program; in practice we never compute the circuit representation and work directly on the source code.

The key ideas relevant to the transposition of linear programs are in [2, Chap. 13] and [1], but they have their roots in the method of the adjoint code for automatic differentiation, a survey can be found in [10]. We first discuss transposition of functions with no const arguments, then go to the general case.

Consider the program in figure 4 and assume that \(h\) and \(g\) have an unique signature where any argument is linear. The transposition is obtained by reversing the flow and transposing the code line by line. When transposing function calls, one simply replaces the function by its transpose. Also, following definition 6, additions become duplications of variables and double uses of variables (such as \(c\)) become additions. It is interesting to notice that optimizing the transposed program by sharing the double assignment to \(\text{trans } h(a)\) and transposing again yields an equivalent improvement to the original program.

68
def (R a)f(R b, R c):
    x, y = g(b, c)
    a = h(x) + h(c) + y

def (R b, R c)fT(R a):
    y, x, c = a, trans h(a), trans h(a)
    b, tmp = trans g(x, y)
    c += tmp

Figure 4: A program with no const variables and its transpose.

Handling const variables permits to treat if statements and products. In if statements each branch is transposed as above; this ultimately permits to treat recursive functions: in fact they are treated no differently than normal functions, as in figure 5.

def (M a)f(linear M b, n):
    if n > 0:
        a = f(b, n - 1)
        a[n] += R.Z(n) * b[n]

def (linear M b)fT(M a, n):
    if n > 0:
        b[n] += R.Z(n) * a[n]
        b += trans f(a, n - 1)

Figure 5: A recursive program and its transpose.

Observe however that this reversal of code may lead to the situation where a function needs a const argument that has not been computed yet. In automatic differentiation, the same problem appears when applying the reverse mode: in this case a forward sweep is needed to precompute the Jacobians of all the functions at the point of differentiation, then a reverse sweep runs through the code in reverse order applying the transpose of the Jacobian to the input vector; see [11, 10]. In our case all the function calls are linear, thus we do not need to compute the Jacobians; but we apply the same technique to predict the values of const variables.

A pathological example is shown in figure 6. Here y is a const variable and its value is needed in order to compute x in the reverse sweep; but the value is only computed by a call to f or its transpose, thus it can only be known too late in the reverse sweep. The forward sweep permits to compute the value of y before it is needed.

Notice however that combining forward sweeps and recursion has a disruptive effect: the transposed algorithm contains now two recursive calls and its complexity is much worse than the original algorithm. The solution is to compute in the forward sweep only the values that are needed; and to compute them only once through a lazy approach. In practice, based on the fact that const outputs only depend on const inputs, transalpyne generates a constification for each signature of each function by stripping out all the linear variables and by replacing function calls with constified function calls. Each time a constified function is evaluated, its output is stored in a memoization table and any future call on the same input values will use the values stored in the table. This technique permits to guarantee that the time complexity of the transposed function obeys the transposition theorem, but the space complexity is potentially increased to be as large as time complexity. This is analogous to what happens in the reverse mode of automatic differentiation; also notice that a technique similar to [11] could be applied in order to obtain a tradeoff between the increases in time and space complexities.

In practice, well written programs will contain few assignments to const variables and pathological functions, such as the one of figure 6, will be rare. For this reason transalpyne only implements the lazy approach in the interpreter, while the compiler produces classical code with a forward and a reverse sweep. For the same reason, we did not implement the technique of [11].

69
def (R a, R b)f(R c, R d):
    if d > R.zero():
        x, y = f(c, d - R.one())
        a, b = x * y, y + R.one()
    else:
        a, b = c, d

def (R c, R b)fT(R a, R d):
    # Forward sweep
    if (d > R.zero()):
        x, y = f(a, d - R.one())
        b = y + R.one()
    else:
        b = d
    # Reverse sweep
    if (d > R.zero()):
        x = a * y
        c, y = trans f(x, d - R.one())
    else:
        c = a

Figure 6: A pathological example and its transpose (relative to the signature \{linear R, const R\}f\{linear R, const R\}) using a forward sweep.

Finally, transalpyne handles for loops by translating them into a tail-recursive function and then transposing it. The resulting transposed function is head-recursive and can be transformed back to a for loop, unless it contains a forward sweep, which happens whenever the loop contains assignments of const variables. This is more powerful than the setting of [1] where for loops can always be transposed to for loops.

7 Conclusion

We presented transalpyne a scripting language that is well suited to implement linear algebra algorithms. transalpyne has no execution capabilities, but permits to define libraries of (multi-)linear transformations that can be used by a target language, this makes it very useful as a scripting language on top of computer algebra systems. transalpyne can be easily integrated in any python-based system: all the user has to do is to make sure its ring elements implement the interface given in Section 3.3.3, then any function written in transalpyne is transposed on the fly by the interpreter and can be called by other functions written in python. As more output languages will be added to transalpyne’s compiler, integration will be possible with other computer algebra systems or generic user written code, although this requires some more effort by the user.

The main features of transalpyne are its ability to discover linearizations of computer programs and to transpose linear programs. The result of the transposition is almost as time and space efficient as the original program, this permits to quickly and automatically implement pairs of algorithms related by duality that are found in the literature and that required a lot of hard man-work to be derived. Some examples we have in mind are the power projection and the middle product that so often recur in algebraic algorithms [22, 19, 12]. Having ourselves spent a few weeks deriving and implementing transposed algorithms for [4], we can testify on how useful transalpyne would have been at that time!

Hence, we believe that transalpyne will prove itself as a useful tool to any computer algebraist. transalpyne is open source software released under the CeCILL license\(^4\). We are planning to release the first stable version in the next few months, it will be available at http://transalpyne.gforge.inria.fr/.

\(^4\)http://www.cecill.info/
References


71
Abstracts of Recent Doctoral Dissertations in Computer Algebra

Communicated by Jeremy Johnson

Each month we are pleased to present abstracts of recent doctoral dissertations in Computer Algebra and Symbolic Computation. We encourage all recent Ph.D. graduates (and their supervisors), who have defended in the past two years, to submit their abstracts for publication in CCA.

Please send abstracts to the CCA editors <editors_SIGSAM@acm.org> for consideration.

Author: Stanislav Bulygin
Institution: Department of Mathematics, University of Kaiserslautern, Germany
Thesis Advisor: Gert-Martin Greuel
Committee Members: Ulrich Dempwolff (chair), Gert-Martin Greuel, Gerhard Pfister

Title: Polynomial system solving for decoding linear codes and algebraic cryptanalysis
Parametric Polynomial System Discussion: Canonical Comprehensive

This book that represents the author’s Ph.D. thesis is devoted to applying symbolic methods to the problems of decoding linear codes and of algebraic cryptanalysis. The paradigm we employ here is as follows. We reformulate the initial problem in terms of systems of polynomial equations over a finite field. The solution(s) of such systems should yield a way to solve the initial problem. Our main tools for handling polynomials and polynomial systems in such a paradigm is the technique of Gröbner bases and normal form reductions.

The first part of the book is devoted to formulating and solving specific polynomial systems that reduce the problem of decoding linear codes to the problem of polynomial system solving. We analyze the existing methods (mainly for the cyclic codes) and propose an original method for arbitrary linear codes that in some sense generalizes the Newton identities method widely known for cyclic codes. We investigate the structure of the underlying ideals and show how one can solve the decoding problem – both the so-called bounded decoding and more general nearest codeword decoding – by finding reduced Gröbner bases of these ideals. The main feature of the method is that unlike usual methods based on Gröbner bases for “finite field” situations, we do not add the so-called field equations. This tremendously simplifies the underlying ideals, thus making feasible working with quite large parameters of codes. Further we address complexity issues, by giving some insight to the Macaulay matrix of the underlying systems. By making a series of assumptions we are able to provide an upper bound for the complexity coefficient of our method. We address also finding the minimum distance and the weight distribution. We provide solid experimental material and comparisons with some of the existing methods in this area.

In the second part we deal with the algebraic cryptanalysis of block iterative ciphers. Namely, we analyze the small-scale variants of the Advanced Encryption Standard (AES), which is a widely used modern block cipher. Here a cryptanalyst composes the polynomial systems which solutions should yield a secret key used by communicating parties in a symmetric cryptosystem. We analyze the systems formulated by researchers for the algebraic cryptanalysis, and identify the problem that conventional systems have many auxiliary variables that are not actually needed for the key recovery. Moreover, having many such auxiliary variables, specific to a given plaintext/ciphertext pair, complicates the use of several pairs which is common in cryptanalysis. We thus provide a new system where the auxiliary variables are eliminated via normal form reductions. The resulting system in key-variables only is then solved. We present experimental evidence that such an approach is quite good for small scaled ciphers. We investigate further our approach and employ the so-called meet-in-the-middle principle to see how far one can go in analyzing just 2–3 rounds of scaled ciphers. Additional “tuning techniques” are discussed together with experimental material. Overall, we believe that the material of this part of the thesis makes a step further in algebraic cryptanalysis of block ciphers.

A short description of the thesis is on the KLUDO webpage: http://kluedo.ub.uni-kl.de/volltexte/2009/2350/
The methods of Scientific Computing play an important role in the natural sciences and engineering. Significance and impact of computer algebra methods and computer algebra systems for scientific computing has increased considerably over the last decade. Nowadays, computer algebra systems such as Maple, Magma, Mathematica, MuPAD, Singular and others enable their users to exploit their powerful facilities in:

- symbolic manipulation
- numerical computation
- visualization

The ongoing development of computer algebra systems, including their integration and adaptation to modern software environments, puts them to the forefront in scientific computing and enables the practical solution of many complex applied problems in the domains of natural sciences and engineering.

The topics addressed in the workshop cover all the basic areas of scientific computing as they benefit from the application of computer algebra methods and software:

- exact and approximate computation
- numerical simulation using computer algebra systems
- parallel symbolic-numeric computation
- problem-solving environments
- internet accessible symbolic and numeric computation
- symbolic-numeric methods for differential, differential-algebraic and difference equations
- algebraic methods in geometric modeling
- algebraic methods for nonlinear polynomial equations and inequalities
- symbolic and numerical computation in systems engineering and modeling
- algorithmic and complexity considerations in computer algebra
- computer algebra in industry
- computer algebra in nanotechnology
- solving problems in natural sciences
- automated reasoning in algebra and geometry

Program Committee. Wolfram Koepf (chair, Kassel), Sergei Abramov (Moscow), Alkis Akritas (Volos), Gerd Baumann (Cairo), Hans-Joachim Bungartz (Munich), Andreas Dolzmann (Saarbrücken), Victor F. Edneral (Moscow), Ioannis Z. Emiris (Athens), Jaime Gutierrez (Santander), Richard Liska (Prague), Marc Moreno Maza (London, Canada), Markus Rosenkranz (Canterbury), Mohab Safey El Din (Paris), Yosuke Sato (Tokyo), Werner M. Seiler (Kassel), Doru Stefanescu (Bucharest), Serguei P. Tsarev (Krasnoyarsk), Evgenii V. Vorozhtsov (vice chair, Novosibirsk), Andreas Weber (Bonn), Eva Zerz (Aachen).

Important Dates.

- Deadline for submission: April 19, 2010
- Notification of acceptance: May 23, 2010
- Deadline for final version: June 6, 2010
4th International Workshop on
Differential Algebra and Related Topics (DART IV)
Beijing, China
27–30 October 2010
http://mmrc.iss.ac.cn/~dart4

Based on the foundational works of Ritt and Kolchin since the 1930s, differential algebra has evolved into an extremely rich subject during the last two decades. Differential Algebra and Related Topics (DART) is a series of workshops which offer an opportunity for participants to present original research, to learn of research progress and new developments, and to exchange ideas and views on differential algebra and related topics. DART-IV is the fourth in this series. The previous three DARTs were held at Rutgers University at Newark, NJ (USA) in 2000, 2007 and 2008.


Invited speakers (confirmed):
- Chengming Bai (Nankai Univ.)
- Gloria Mari Beffa (Univ. Wisconsin-Madison)
- Yuqun Chen (South China Normal Univ.)
- Lucia Di Vizio (Univ. Paris 7)
- Charlotte Hardouin (IWR, Univ. Heidelberg)
- Mark van Hoeij (Florida State Univ.)
- Irina Kogan (North Carolina State Univ.)
- Francois Lemaire (Univ. Lille 1, France)
- Alexander Levin (Catholic Univ. America)
- Jean Levine (Ecole des Mines de Paris)
- Fang Li (Zhejing Univ.)
- Jorge Martin-Morales (Univ. Zaragoza)
- Alexander V. Mikhailov (Univ. Leeds)
- Ned Nedialkov (McMaster Univ.)
- Alexei Ovchinnikov (City Univ. New York)
- Daniel Robertz (RWTH Aachen Univ.)
- Ravi Srinivasan (Rutgers Univ.)
- Sebastian Walcher (RWTH Aachen Univ.)
- Chunming Yuan (KLMM, CAS)
- Jean Levine (Ecole des Mines de Paris)

Presentations. In addition to 50-minute and 25-minute talks, there will be sessions for poster presentations. If you would like to present a poster, please submit an abstract in PDF format of no more than two pages by September 1, 2010. Submission is via EasyChair at the web site http://www.easychair.org/conferences/?conf=dart4. The workshop will invite a limited number of the poster presenters to give 25-minute talks, and cover their local expenses (hotel rooms and registration fees). A decision on posters will be made by September 15, 2010.

Registration. See http://www.mmrc.iss.ac.cn/~dart4/registration.html.

Organizers.
- Workshop Chair: Xiao-Shan Gao
- Program Committee: Li Guo, Julia Hartmann, Evelyne Hubert, Ziming Li, Francois Ollivier, Michael Singer, William Sit
- Local Arrangements: Ruyong Feng, Chunming Yuan
3rd Workshop on Compact Computer Algebra  
at CICM 2010  
Paris, France  
6 July, 2010  
http://www.orcca.on.ca/conferences/cca2010/  

Even though compact design is no longer a vital necessity for main-stream computer algebra systems, it is a central question in emerging settings. Compact systems are important for hand-held devices, embedded computer algebra modules (e.g. for smart document processors) and web-based computing to name a few areas. Additionally, compact data representations can be essential when dealing with very large problems.

The aim of the CCA events is to communicate the ideas supporting the subject of “compactness” in algorithms, data organization and system design for computer algebra.

Continuing the series of CCA meetings previously held in Linz, Austria (2008) and in Grand Bend, Canada (2009), the 3rd workshop on Compact Computer Algebra will take place in Paris, France as a part of CICM 2010 conference.

**Workshop Scope:** the whole spectrum of issues of compact CA including, but not limited to:

- space-efficient data structures
- memory-efficient implementations
- compact kernels
- math education tools
- portable and Internet-accessible symbolic calculators
- CAS for personal digital assistants
- “spell checkers” for math content in document processing software
- validators for online and offline mathematical recognizers
- backend engines to pen-computing interfaces
- math editing components for 2D expression

**Session organizers**

- Elena Smirnova / Texas Instruments / Education Technology, USA  
- Stephen M. Watt / Ontario Research Centre for Computer Algebra / University of Western Ontario, Canada  
- Mitsushi Fujimoto / Department of Mathematics / Fukuoka University of Education, Japan

**Web resources**

- CICM 2010 home page:  
  http://cicm2010.cnam.fr/  
- 3rd Workshop on Compact Computer Algebra web site:  
  http://www.orcca.on.ca/conferences/cca2010  
- Call for Papers and Software demos for Compact Computer Algebra page:  
  http://www.orcca.on.ca/conferences/cca2010/submissions.html
17th International Conference on
Logic for Programming, Artificial Intelligence and Reasoning
Yogyakarta, Indonesia
10–15 October, 2010
http://www.computational-logic.org/lpar-17/Home.html

The series of International Conferences on Logic for Programming, Artificial Intelligence and Reasoning (LPAR) is a forum where, year after year, some of the most renowned researchers in the areas of logic, automated reasoning, computational logic, programming languages and their applications come to present cutting-edge results, to discuss advances in these fields, and to exchange ideas in a scientifically emerging part of the world. The 17th LPAR will be held in Yogyakarta, Indonesia.

Conference Chair: Steffen Hoelldobler
Programme Chairs: Chris Fermueller, Andrei Voronkov

Submissions of two kinds are welcome:

- Regular papers that describe solid new research results.
- Experimental and tool papers that describe implementations of systems, report experiments with implemented systems, or compare implemented systems.

See the web site http://www.computational-logic.org/lpar-17/Home.html for all the details.

LPAR-17 WORKSHOPS October 10th, 2010

APS 5 - 5th International Workshop on Analytic Proof Systems

Analyticity is a topic that connects foundational issues in logic with applications, mainly in automated deduction and analysis of proofs. The workshop is primarily intended to enhance awareness for its topic and to promote corresponding discussions and contacts between experienced experts and younger colleagues. The submission deadline is 10th September.

Organizers: Matthias Baaz, Christian Fermueller
http://www.logic.at/staff/chrisf/ws/LPAR-AS-5.html

IWIL 2010 - The 8th International Workshop on the Implementation of Logics

IWIL has been unusually successful in bringing together many talented developers, and thus in sharing information about successful implementation techniques for automated reasoning systems and similar programs. We are looking for contributions describing implementation techniques for and implementations of automated reasoning programs, theorem provers for various logics, logic programming systems, and related technologies. The submission deadline is 9th August.

Organizers: Evgenia Ternovska, Stephan Schulz, Geoff Sutcliffe