Σ^{IT} – A strongly-typed embeddable computer algebra library

Manuel Bronstein

Institute for Scientific Computation
ETH Zentrum IFW
CH-8092 Zürich, Switzerland

Abstract. We describe the new computer algebra library Σ^{IT} and its underlying design. The development of Σ^{IT} is motivated by the need to provide highly efficient implementations of key algorithms for linear ordinary differential and (q)-difference equations to scientific programmers and to computer algebra system users, regardless of the programming language or interactive system they use. As such, Σ^{IT} is not a computer algebra system per se, but a library (or substrate) which is designed to be "plugged" with minimal efforts into different types of client applications.

1 Introduction

Σ^{IT} is a new computer algebra library under development at the ETH Zürich. It is not designed to evolve a general purpose computer algebra system, but to the contrary, it is designed to provide reusable and efficient implementations of highly specialized primitives and algorithms for linear ordinary differential and (q)-difference equations, either to any other computer algebra systems, or to scientific programmers coding in more traditional compiled languages. The need for providing such primitives was recognized by the EC CATHODE working group, whose solution to this problem was to provide primitives with compatible naming schemes and identical semantics written in all of `axiom`, `maple` and `reduce`. This allows users of those primitives to program higher level algorithms, e.g. the Kovacic algorithm for second order linear ordinary differential equations [6], on top of those primitives in a portable way accross those three systems. Our approach is to provide a single, compiled version of those algorithms, and make it both linkable into applications, or callable from any interactive shell. This single-source approach allows for easier maintenance, tuning, and integration of future algorithmic improvements. Thus, our emphasis is not on wide coverage of algebra, but on providing selected state of the art algorithms in a "plug-and-play" fashion to a wide range of different client applications. This is what we mean by the term "embeddable" when it appears in this paper. The closest previous attempt to provide such services to the scientific programming community is the Weyl computer algebra substrate [14], which can be seen as a set of algebraic datatypes that can be embedded into a Lisp system, providing it with computer algebra services. We describe in this paper the features of Σ^{IT} that make it significantly different from other libraries, namely:

- its use of the Aldor[1] programming language [12] and of its OOP features;
- its emphasis on skew-polynomials and the other ground mathematical objects of pseudo-linear algebra [4];
- its smooth merging of both a strongly-typed object-oriented and functional interface for low-level linking, and a typeless string-based interface for high-level process to process communication. This efficient merging is the key to efficient embeddability at all levels without type inference.

2 The use of Aldor

Σ^{IT} is entirely written in the Aldor programming language [12]. Aldor is a new compiled language, which can be described as a second generation successor of the language used to program the axiom library [5]. We refer to [12, 13] for more details about Aldor. Σ^{IT} has been designed from the start to take advantage of the following language features, whose benefits for programming algebraic algorithms are well known:

- Object-oriented programming with templating, subclassing and multiple inheritance. This allows for generic algorithms over mathematical categories.
- Aldor functions produce C-linkable object code. This allows for linking together Aldor modules with C, C++ and Fortran libraries.
- Aldor provides an abstraction level similar to axiom, so there is no need to use two different languages, one for the kernel (typically C/C++ for efficiency) and one for the high level library code (typically interpreted).
- Similarly to Lisp, Aldor provides built-in big integers, transparent dynamic allocation and a garbage collector, freeing the programmer from such tasks as memory management.

We illustrate the high-level of mathematical abstraction provided by Aldor and Σ^{IT} in Section 4, where we describe the use of Σ^{IT} within Aldor programs. A side-benefit of using Aldor, is the use of the asdoc LaTeX2e document class [7], which has been developped specially for the purpose of processing documentation in Aldor source files. Thanks to this class, each Σ^{IT} exported type and function is documented in the source code, producing a reference manual with a unix-like manual page for each exported name.

3 The algebraic facilities

As explained in the introduction, the goal is to provide primitives and algorithms for the manipulations of skew-polynomials [4]. Those objects form a common abstraction of linear ordinary differential and (q)-difference operators, so for example the Weyl algebra $\mathbb{Q}[x, d/dx]$ is a particular skew-polynomial ring. Since those rings are noncommutative, the basic category hierarchy provided by Σ^{IT} includes several noncommutative classes (Fig. 1). It was chosen to provide only categories

[1] Aldor is mostly known under its internal working names $A^{\#}$ and AXIOM-XL

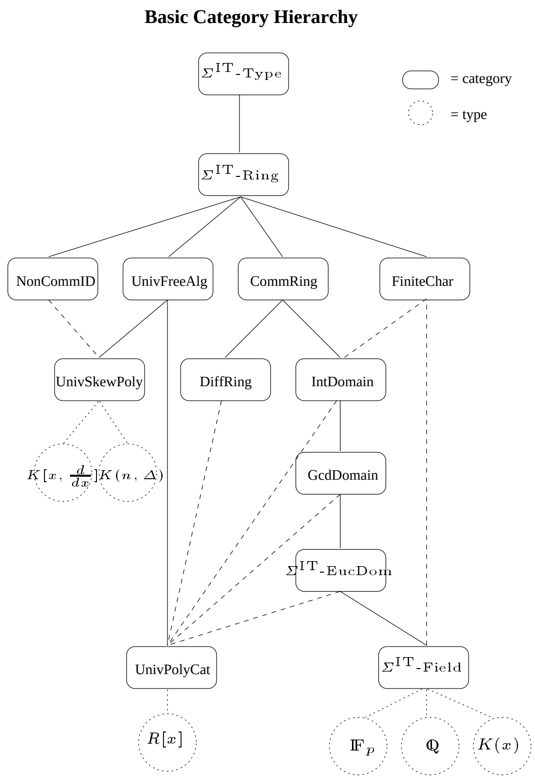

Fig. 1. The Σ^{IT} basic category hierarchy

for which there are existing types and implemented generic algorithms in the library, rather than to attempt as wide a coverage as in the `axiom` category hierarchy [5]. The resulting category structure is a rooted DAG. Σ^{IT} provides implementations of all the CATHODE primitives for arbitrary skew-polynomials, *i.e.*:

- ring arithmetic, plus left and right Euclidean division;
- left and right greatest common divisors and least common multiples;

In addition, adjoints and symmetric powers [9] are provided for differential operators. The only significant higher-level operation provided for a differential operator L is the computation of its rational kernel, i.e. $\ker(L) \cap \mathbb{Q}(x)$. This operation is a key primitive of differential Galois theory [11], and by using polyalgorithms and recent improvements [1], our implementation of rational kernel is significantly faster than the existing alternatives that are distributed with commercial computer algebra. For example, on a single-processor SPARC 10/41, Σ^{IT} was able to compute the symmetric square of

$$L = \frac{d^7}{dx^7} - x\,\frac{d}{dx} + \frac{1}{2} \tag{1}$$

and its one-dimensional rational kernel in 3 minutes of CPU time. The same computation took 3.5 hours of CPU using `maple` together with its `ratlode` share-library package, and did not terminate after 12 hours using the `ratDsolve` `axiom` library function. One of the reasons is that different algorithms for this problem have a complexity that is cubic either on the size of the operator, or on the size of its solutions, so only a polyalgorithm making the right choice between the undetermined coefficient method and the recurrence method of [1] can be efficient on all inputs. An example of a small equation with large solution is the legendre operator

$$L_n = (1-x^2)\frac{d^2}{dx^2} - 2x\,\frac{d}{dx} + n(n+1)\,,$$

which has a one-dimensional rational kernel. For $n = 200$, $\Sigma^{\rm IT}$ took 3.8 CPU seconds vs. 6.3 seconds for the `maple` `ratlode` package, while for $n = 500$ this ratio becomes 18.6 to 40. The single-source approach makes algorithmic improvements such as [1, 2, 3] available at the same time to all computer algebra systems, with no porting of code necessary.

The more traditional objects of commutative algebra, e.g. univariate polynomial rings are also provided in $\Sigma^{\rm IT}$, together with a standard set of basic algorithms, e.g. modular gcd's, polynomial remainder sequences, resultants, and factorization in $\mathbb{F}_p[x]$, $\mathbb{Z}[x]$ and $\mathbb{Q}[x]$. Types for matrices and vectors are also included. All the relevant types for polynomials, operators and linear algebra are provided in both dense and sparse versions. As of January 96, the current count of exported operations is over 100 type constructors, exporting over 500 different names (each name, e.g. `+` corresponds to many different functions).

4 Low-level embeddability

By low-level embeddability, we mean the use of $\Sigma^{\rm IT}$ as a strongly-typed library to be linked in an application program. As for any other compiled language, $\Sigma^{\rm IT}$ is provided as an archive of object files, together with one include file `sumit.as`, and including it into any `Aldor` program gives access to the full $\Sigma^{\rm IT}$ type system and its algorithms. Since $\Sigma^{\rm IT}$ is a compatible extension of the `axllib` `Aldor` standard library, one just needs to replace the `#include "axllib"` line by `#include "sumit"`. The following sample program, which computes the rational kernel of the symmetric square of the operator (1) illustrates the abstraction level provided by `Aldor` and $\Sigma^{\rm IT}$. Note the use of calls to `monom` to create the independent variable and derivation, and the required coercion of the rational number 1/2 to a differential operator. Those strong-typing "annoyances" for non-programming users will not appear in the typeless interface presented in the next section.

```
----------------------- Katz' G2 example ------------------------
#include "sumit"
macro {
     Z   == Integer;                            Q  == Quotient Z;
     QX  == DenseUnivariatePolynomial(Q, "x"); Qx == Quotient QX;
     QXD == LinearOrdinaryDifferentialOperator(QX, "D");
}
main():() == {
     import from SingleInteger, Z, Q, QX, Qx, QXD;
     import from Vector Qx, LODORationalSolutions(QX, QXD);
     x:QX   := monom;               -- create independent variable
     dx:QXD := monom;               -- create derivation
     L := dx^7 - x*dx + (1/2)@Q::QX::QXD;
     V := rationalKernel symmetricPower(L, 2);
     print<<#V<<"-dimensional space of solutions found:"<<newline;
     for i in 1..#V repeat print << V.i << newline;
}
main();
```

Since `Aldor` is a general purpose programming language, Σ^{IT} can be used as a computer algebra engine inside programs whose main emphasis is not computer algebra. In fact, the only published research use of Σ^{IT} to date is of this form: it was used as a skew-polynomial engine inside a symbolic-numeric implementation of Balser's multisummation algorithm for divergent series [8]. The resulting program is a symbolic-numeric software that links in addition to a C-graphics library for rendering its results. The linked use of Σ^{IT} is of course its most flexible and efficient use.

A direct benefit of the use of the `Aldor` language is that Σ^{IT} can be linked in the above fashion to C/C++ and Fortran programs, through compiled stubs for each function used. Those compiled stubs only need to convert the arguments between C and `Aldor` types, so binary data can still be shared. Here is an example of a stub exporting polynomial factorization over $\mathbb{F}_p$ to C/C++ programs. Note the `export` statement making its arguments available to client programs, as well as the storage of the coefficient prime field into the POLY structure, which is itself exported to client programs.

```
----------------------------- dupfp.as --------------------------
-- C interface to dense Fp[X] factorization
#include "sumit"
macro {
    Z      == SingleInteger;     DUP == DenseUnivariatePolynomial;
    POLY   == Record(F:PrimeFieldCategory, Fx:DUP F);
    FACTOR == Record(factor:POLY, exponent:Z);
    FPOLY  == Record(lc:Z, size:Z, factors:PrimitiveArray FACTOR);
}
export {
    FpXfactor: POLY -> FPOLY;
```

```
    FpXmake:   (Z, PrimitiveArray Z, Z) -> POLY;
    FpXprint:  (POLY, String) -> ();
} to Foreign C;

FpXprint(p:POLY,x:String):() == print0(explode p)(x);
local print0(Fp:PrimeFieldCategory,p:DUP Fp)(x:String):()==print(p,x);

FpXmake(deg:Z, c:PrimitiveArray Z, charac:Z):POLY == {
    macro Fp == SmallPrimeField charac;
    import from Integer, Fp, DUP Fp;
    p:DUP Fp := 0;
    for i in 0..deg repeat p:=add!(p,monomial(c(i+1)::Fp,i::Integer));
    [Fp, p];
}

FpXfactor(p:POLY):FPOLY == factor0 explode p;
local factor0(Fp:PrimeFieldCategory, p:DUP Fp):FPOLY == {
    import from Z, Product DUP Fp, FACTOR, PrimitiveArray FACTOR;
    import from PrimeFieldUnivariateFactorizer(Fp, DUP Fp);
    import from Integer, Record(lcoeff:Fp, factors:Product DUP Fp);
    r := factor p;    v:PrimitiveArray FACTOR := new #(r.factors);
    for term in r.factors for i in 1@Z.. repeat {
        (f, n) := term;   v.i := [[Fp, f], retract n];
    }
    [lift(r.lcoeff), #(r.factors), v];
}
```

To the above stubs corresponds the C header file `sumit_dupfp.h`, which declares the 3 functions that are exported from the stub, but using C types as arguments:

```
/* sumit_dupfp.h: header for sumit interface to Fp[x] factorizer */
typedef void *FPX;

typedef struct single_factor {
    FPX factor;
    unsigned int exponent;
} FACTOR;

typedef struct factored_poly {
    int leadingCoefficient;
    unsigned int numberOfFactors;
    FACTOR **factors;
} FACTORED;

extern FPX  FpXmake(unsigned int deg, int *coeffs, unsigned int charac);
extern void FpXprint(FPX poly, char *varname);
extern FACTORED *FpXfactor(FPX poly);
```

The above header file is the only visible part of the stub for client programs. For example, the following C program computes the factorization of $x^7 + 5x^6 + 7x^5 + 9x^4 + 3x^3 + 4x^2 + x + 6$ in $\mathbb{F}_{11}[x]$:

```
#include <stdio.h>
#include "sumit_dupfp.h"
main() {        /* x^7+5x^6+7x^5+9x^4+3x^3+4x^2+x+6*/
    int i, coeff[8] = {6, 1, 4, 3, 9, 7, 5, 1};
    FPX p = FpXmake(7, coeff, 11);   /* F_11[x] */
    FACTORED *fp = FpXfactor(p);
    printf("The factorization of $");
    FpXprint(p, "x");
    printf("$ in $F_11[x]$ is: $$\n %d ", fp->leadingCoefficient);
    for (i = 0; i < fp->numberOfFactors; i++) {
        printf("(");
        FpXprint(fp->factors[i]->factor, "x");
        printf(")^%d ", fp->factors[i]->exponent);
    }
    printf("\n$$\n");
}
```

Embedding Σ^{IT} into Fortran programs is done in a similar fashion, with stubs and `export` statements. Because the `Aldor` compiler can produce Lisp code, it is also possible to load the entire Σ^{IT} type system into a Lisp system, for example into `axiom`. Although the linked use of Σ^{IT} into other compiled languages is less flexible than into `Aldor` programs, this makes computer algebra algorithms available with minimal effort to general purpose programs. Such computer algebra services are already being provided by the many C-based computer algebra kernels available (e.g. PARI, SACLIB, GB, KANT, SIMATH, GANITH, KAN), but none of them provides algorithms for linear operators, with the exception of KAN [10], which provides however a different algorithm (Gröbner bases).

5 High-level embeddability

The previous section described how Σ^{IT} can be used as an engine by programmers. Since most computer algebra system systems provide a higher level style of programming in their interface language, Σ^{IT} is also designed to be used transparently from almost any interactive shell. The main problem to solve for such interaction is the strong typing of all Σ^{IT} objects. For example, the `axiom` interpreter [5] provides an interface between a mostly typeless user world and a strongly-typed algebra library, but with the heavy cost of type-inferencing. A different design concept was selected for Σ^{IT}: the use of light-weight servers providing typeless access to selected functionalities (in contrast to the `axiom` interpreter, which provides access to all the library). In some sense, servers can be seen as string-based stubs for process to process communication. Since the set of provided services is fixed once a server is compiled, it is crucial that designing new servers or extending existing ones can be done very easily. This is made possible by the object-oriented design of `Aldor`: generic subclassable parsers and interpreters are provided, so new servers with additional or different semantics can be derived easily from existing ones. We describe this process in more details in this section.

5.1 Expression Trees

In order to provide a typeless face to the outside world, an expression tree type is implemented in Σ^{IT}. A significant difference with Lisp S-expressions, or the parse and output trees provided by most systems, is that the operators (internal nodes) in the Σ^{IT} expression trees are full-fledged types (classes) rather than names or placeholders. The common category (abstract class) for all the operators enforces the export of various methods for communication with the outside world (e.g. conversion to TEX, `axiom` or `maple` input, generation of C code). Should a new conversion be desired to a new format or protocol (e.g. OpenMath), it is sufficient to add a corresponding method to each operator, without modifying any of the types in the algebra engine. In several cases, adding a default implementation to the operator category is enough (for example most prefix operators generate TEX input by adding a \ in front of their name). The root category in the algebra hierarchy (Fig. 1) enforces a conversion into expression trees, so every exported type is convertible to a typeless tree, thereby allowing its values to be exported to the outside world. For the reverse direction, Σ^{IT} provides an expression parser that produces expression trees from any input stream. This results in a two-sided design (Fig. 2), with a strongly-typed algebra side and a typeless expression tree side that shields the algebra completely from the outside world. It is possible

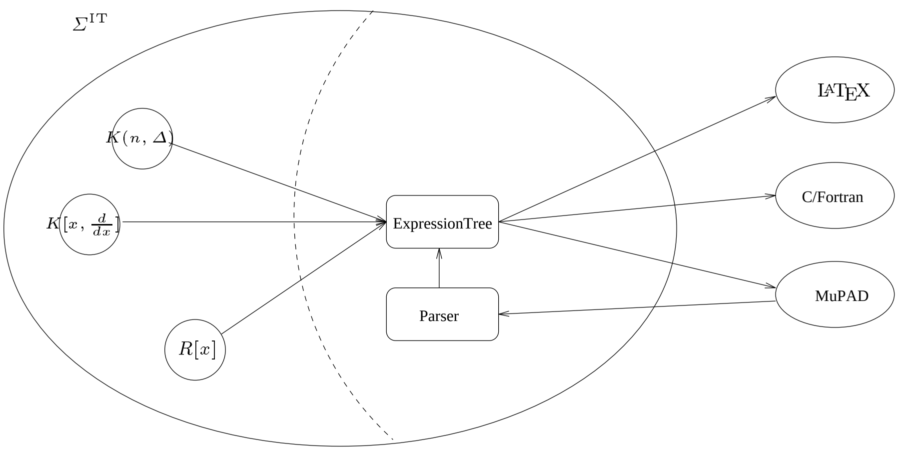

Fig. 2. The strongly-typed/typeless 2-sided design

to program mathematical algorithms operating on the typeless expression trees (e.g. numerical evaluation or pattern matching), and to take advantage of their object-orientedness in the process. In contrast to several systems, Σ^{IT} uses the same tree type for both input and output.

5.2 Partial rings

The only remaining problem in order to complete a server is the interpreting of an expression tree, i.e. converting an expression tree produced by the parser into a strongly-typed object. The semantics of this operation depend of course on the target type, and cannot be carried out without type-inference in the absence of information. In order to avoid the cost of type-inference, a server provides access only to a limited set of types. To support such a set of types, Σ^{IT} provides the category `PartialRing`, which is an abstract class for generic unions of rings. Any partial ring (∂-ring for short) must export at least all the ring and field arithmetic $(0, 1, +, -, *, /, \hat{})$ but all the exported operations are allowed to fail on some combinations of inputs, for example if the operands are not in compatible components of the union. Failure is reported through the use of the `Partial` type constructor of `Aldor`, although raising an exception will also be a possibility when exceptions are supported by `Aldor`. An implemented example is the `bernina` server for linear ordinary differential operators. This server exports to the outside world the following ∂-ring:

$$\mathbb{Z} \cup \mathbb{Q} \cup \mathbb{Q}[x] \cup \mathbb{Q}(x) \cup \mathbb{Q}[x, \frac{d}{dx}] \cup \mathbb{Q}(x, \frac{d}{dx}) \cup \mathbb{Q}(x)^{\mathbb{N}} \tag{2}$$

where $\mathbb{Q}(x)^{\mathbb{N}}$ means vectors of fractions of arbitrary length. Because of the various natural embeddings, the above ∂-ring is implemented as the 2-branch union $\mathbb{Q}(x, d/dx) \cup \mathbb{Q}(x)^{\mathbb{N}}$, but the knowledge of all other components is used when deciding whether an operation (e.g. exponentiation) can be carried out. Note that a ∂-ring exports only signatures involving itself (% in `Aldor`), so the components of its union are completely hidden from the outside. Each exported operation from a ∂-ring is programmed by checking the components of the arguments and dispatching the call to the appropriate Σ^{IT} types.

5.3 Evaluators

Once we have a specific ∂-ring R in mind, the need for type-inferencing is removed, since the interpreter only needs to convert expression trees to R. Since R is expected in addition to export operations corresponding to all the functions exported by the server under consideration, the interpreter only needs to traverse the tree, calling the appropriate function from R at each node. To make this process easy, Σ^{IT} provides a category `Evaluator(R:PartialRing)`, which can be seen as an abstract interpreter. But since any ∂-ring must provide ring and field arithmetic as well as an embedding from $\mathbb{Z}$ into itself, the default generic interpreter is able to process correctly all the integer and symbolic leaves, arithmetic operators, as well as assigment operators. Thus, a server only needs to provide function for handling operators not covered above, usually only the exported prefix functionalities. For example, the following code is the complete `bernina` interpreter, where `MultiLodo` is an implementation of the ∂-ring given by (2). All the other interpreter functionalities are provided by the generic behaviour of `Evaluator(R)`.

```
---------------------- evalodo.as -----------------------------
#include "sumit"
#library pring "multlodo.ao"
import from pring;
macro R == MultiLodo(QX, Qxd);

LODOEvaluator(QX:UnivariatePolynomialCategory Quotient Integer,
 Qxd:LinearOrdinaryDifferentialOperatorCategory Quotient QX):
  Evaluator R == add {
    evalPrefix!(name:String, nargs:SingleInteger,
        args:List R, tab:SymbolTable R):Partial R == {
            nargs = 1 => eval1(name, first args);
            nargs = 2 => eval2(name, first args, first rest args);
            failed;
    }
    eval1(name:String, x:R):Partial R == {
        name = "adjoint" => adjoint x;
        name = "diff" => D(x, 1);
        name = "kernel" => kernel x;
        failed;
    }
    eval2(name:String, x:R, y:R):Partial R == {
        name = "apply" => x y;
        name = "diff" => D(x, y);
        name = "symmetricPower" => symmetricPower(x, y);
        name = "kernel" => kernel(x, y);
        failed;
    }
}
```

5.4 Servers

Together with the built-in parser, a combination of a ∂-ring and an evaluator for it can be turned trivially into a read-eval loop that takes its input from a stream, processes it, and writes the result to another stream in any supported format when requested. Tools for standard read-eval loops (timing, prompting, output format conversions) are provided by the `Shell` type, which is itself parametrized by both a ∂-ring and an evaluator for it, allowing for rapid coding of a top-level server. The end-product is either a library or a stand-alone process that provides services while appearing to the outside like a typeless expression-tree

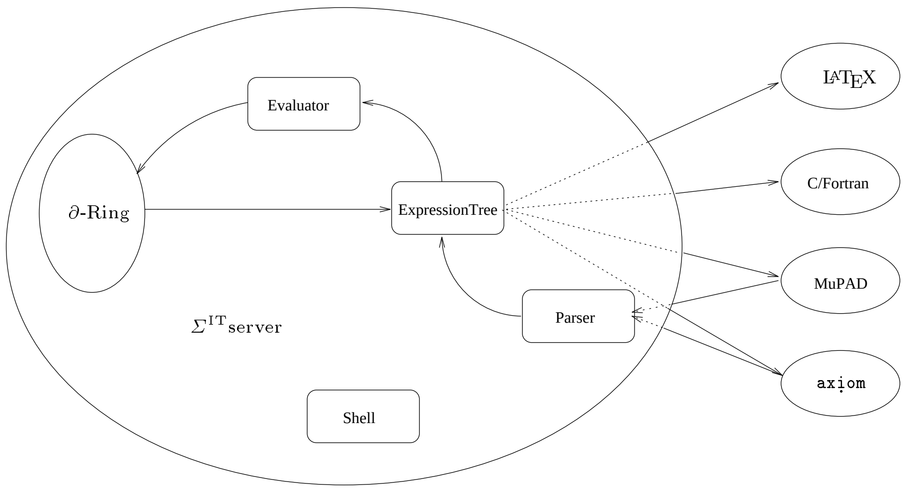

Fig. 3. A typical Σ^{IT} server

based computer algebra server (Fig. 3). Two sample servers are currently implemented, one for linear ordinary differential operators, and one for univariate polynomial factorization. A server can be used in 2 slightly different ways: linked into an application, in which case it provides a single entry-point that parses and execute a command string, or as a stand-alone process communicating via files or sockets. We illustrate the latter use of `bernina` from `maple`: the function below is a `maple` function that computes the kernel of the symmetric square of an operator L given as a polynomial in x and D, by calling a Σ^{IT} service via files:

```
quadraticInvariants := proc(L)
    readlib(write); open(`bernin`);
    writeln(`L := `); writeln(lprint(L)); writeln(`;`);
    writeln(`maple(kernel(symmetricPower(L,2)));`); close();
    open(`bernout`); write(`sumitresult := `); close();
    system(`bernina -q < bernin >> bernout`);
    read bernout; sumitresult; end;
```

Σ^{IT} can then compute the kernel of the symmetric square of the operator (1):

```
> v := quadraticInvariants(D^7 - x*D + 1/2);
> print(v);
                                        [ x ]
```

Note that the above function can be used within further `maple` programs, providing for a short implementation of the completely reducible cases of [11]. Of course, the above function can be easily translated to any interactive shell providing file I/O and linear output. In addition, each server provides a basic terminal interface, and can thus be used interactively directly. We conclude with the above computation performed directly in the `bernina` server. Comparing with the sample Aldor program shown in Section 4 for this computation, highlights the typeless feature of the server interface (neither types nor coercions are necessary):

```
                                                                      /\
      -----  IT                                                      /| \
      \                  B E R N I N A                              / |  \
       \                                                           |   \  \
       /                A Sum^it server                            |    |  \
      /                                                            /     \  \
      -----                                                       /       |  \
                   Use ^D to terminate session                   /       /    \
Independent variable = x
Derivation = D
1 --> L := D^7 - x*D + 1/2;
2 --> L2 := symmetricPower(L,2);
3 --> V := kernel(L2);
4 --> axiom(V);
vector [x]
```

References

1. S. A. Abramov, M. Bronstein & M. Petkovšek (1995): On polynomial solutions of linear operator equations, *Proceedings of ISSAC'95,* ACM Press, 290–296.
2. S. A. Abramov & K. Yu. Kvashenko (1991): Fast Algorithms to Search for the Rational Solutions of Linear Differential Equations with Polynomial Coefficients, *Proceedings of ISSAC'91,* ACM Press, 267–270.
3. M. Bronstein (1992): On Solutions of Linear Ordinary Differential Equations in their Coefficient Field, *Journal of Symbolic Computation* **13,** 413–439.
4. M. Bronstein & M. Petkovšek (1996): An introduction to pseudo–linear algebra, *Theoretical Computer Science* **157**, 3–33.
5. R. D. Jenks & R. S. Sutor (1992): *Axiom – The Scientific Computation System,*
6. J. J. Kovacic (1986): An Algorithm for Solving Second Order Linear Homogeneous Differential Equations, *Journal of Symbolic Computation* **2** 3–43.
7. N. Mannhart (1995): *asdoc* – Aldor *documentation class for LaTeX2e,* ftp://ftp.inf.ethz.ch/org/cathode/asharp/asdoc/
8. F. Naegele (1995): *Autour de quelques équations fonctionnelles analytiques,* Thèse de Mathématiques Appliquées, INPG, Grenoble.
9. M. Singer & F. Ulmer (1993): Galois Groups of Second and Third Order Linear Differential Equations, *Journal of Symbolic Computation* **16,** 9–36.
10. N. Takayama (1992): *Kan,* ftp://ftp.math.s.kobe-u.ac.jp/pub/kan/kan96.tgz
11. F. Ulmer & J. A. Weil (1994): *Note on Kovacic's algorithm,* Prépublication 94-13, IRMAR, Rennes.
12. S. M. Watt & al. (1994): A first report on the $A^{\#}$ compiler, *Proceedings of ISSAC'94,* ACM Press, 25–31.
13. S. M. Watt & al. (1994): *The $A^{\#}$ user's guide,* NAG Ltd.
14. R. Zippel (1990): *The Weyl computer algebra substrate,* Technical Report 90-1077, Dept. of Computer Science, Cornell University.