Risa/Asir – A Computer Algebra System

Masayuki Noro and Taku Takeshima
IIAS-SIS, FUJITSU LABORATORIES LTD.
140 Miyamoto, Numazu-shi, Shizuoka, 410-03 JAPAN
(e-mail:noro@iias.flab.fujitsu.co.jp)

Abstract

A computer algebra system risa/asir consists of a command asir for interactive use and several subroutine libraries which can be used as the parts of other programs. The grammar of the user language of asir is a variant of that of C and asir has a built-in dbx-like debugger. Risa's subroutine libraries include basic arithmetic subroutines, parser, evaluator and storage manager, and each of them can be used individually. This paper describes the characteristics and structure of risa system. We also show some sample programs, usage of the debugger and some timing data of fundamental calculations.

1 Introduction

Risa/asir [11] is a computer algebra system. Risa is the name of the whole system, and asir is a command for doing algebraic manipulations interactively. Like Macsyma, REDUCE, Maple, and Mathematica, risa/asir is a rather conservative computer algebra system classified into *hacker-prototype system* according to R. Fateman's classification category[6]. Nevertheless, it is distinguished from the others in the point that it is clearly oriented to independent cooperation for future scientific computation. It is a set of UNIX libraries. Functions in the libraries are considered as software parts to manipulate algebraic formulae. Thus, external programs can use risa/asir functions simply as C-coded library functions. Conversely, risa/asir rely on external softwares for its weak or lacking facilities, e.g. editing, drawing graphs, numerical computations, etc.

PARI system [2] and SAC2-C system [10] are developed under similar design philosophy. PARI is designed for mainly doing number theoretic computations. Its computational power for bignum and bigfloat is extremely high. Many number theoretic functions are implemented and collected into an UNIX subroutine library. These library functions, as those of risa, can be accessed by programs written in C etc.

ISSAC '92-7/92/CA, USA

PARI, however, does not reclaim garbages automatically. Storage management is a matter of users. We have not yet obtained SAC2-C documentations, and so we cannot refer to the detailed design of SAC2-C. It is also a UNIX subroutine library and it has its own list processing system with a garbage collector. In case of risa, garbage collection is automatically done by risa's storage manager. Moreover, risa's storage allocator is compatible with `malloc()` of UNIX. And hence, use of risa subroutines is as convenient as that of popular numerical subroutines.

Now, we introduce our design goals of risa/asir system.

1. **self-contained system**
2. **high portability**
3. **efficiency and compactness**
4. **powerful user program debugger**
5. **modularity**

For each, we describe its necessity and realization in the current risa system. In this paper we call the set of subroutines to execute fundamental arithmetics in risa **"engine"**.

1.1 Self-contained system

There are many computer algebra systems based on Lisp such as REDUCE and Macsyma. The advantage of Lisp is that the storage management, list processing, bignum

processing are built-in. As a consequence, the performance of Lisp-based systems strongly depends on the Lisp. Furthermore, Lisps, in general, require very large runtime object due to many features unnecessary for computer algebra. For these reasons, we have chosen C for the system implementation language. One big problem arising from this decision is the storage management. Our solution is the use of the garbage collector introduced in [3] for the storage manager of risa. This will be described in detail later.

1.2 Portability

The target machines of risa are UNIX machines. The system implementation language of risa is C. This benefits the portability of risa because C is a standard language in UNIX. Unfortunately, a very small part of the system had to be written in assembly language. For achieving efficiency, double word integer multiplication and division must be written in assembly language because they are not supported by a standard library of C. A small part of the garbage collector must also be written in assembly language, because it has machine (architecture) dependent property. These assembly codes, however, are so small and transplantation of risa is not so difficult.

Recently we implemented risa on `LUNA88K` workstation. The CPU of this machine is Motorola `MC88100`. Our garbage collector had no codes for the CPU, so we had to write all the assembly codes mentioned above. But it took less than 1 week to implement it. Especially we could run the garbage collector in 1 day.

1.3 Efficiency and compactness

Almost all computations in computer algebra require large time and space. If the same algorithm is used, the order of the complexity is thought to be identical. In practice, however, the actual time of a calculation is strongly affected by the implementation of the algorithm. Especially, fundamental parts, such as bignum and polynomial arithmetic, that are executed frequently, must be fairy fast. Risa engine shows sufficiently high efficiency. We show its timings later.

Today, it is not rare that a machine is equipped with more than thirty megabytes of memories. Nevertheless, smaller program is better because paging overhead will be increased for larger programs. The object size of asir, though it varies depending on CPU's (RISC or CISC etc.), is about 500Kbytes on SPARC (statically linked on SunOS4.1.2; comparable figure to Maple). This also enables asir run on machines with little memories.

1.4 Powerful debugger

Computer algebra languages are designed for two kind of objectives, one for interactive use and the other for programming higher level algebraic computations. In the latter context, powerful debugging facility is indispensable. There is a source-level debugger of C named dbx [16], which is one of the standard debuggers in BSD UNIX and which makes debugging of C programs quite easy. We are convinced that mere tracing facility is insufficient for computer algebra programming, especially for system integrators for higher algebraic computation. We implemented a debugger which has a subset of dbx commands. Dbx commands frequently used are all realized, e.g., commands for setting breakpoints at any lines of source programs, examining values of variables and step execution. Its effective debugging potential is satisfactory to us through several programming experiences of higher level algorithms, e. g., factoring univariate polynomials over algebraic number fields by norm method. (See Subsec. 6.2.2.)

1.5 Modularity

We believe that the subroutines for algebraic computation should be compiled as libraries accessible by other programs, just as many numerical subroutine libraries. For this purpose, we clearly separated the parser and the functions for computation of internal forms and made each part available as a UNIX subroutine library. The main obstacle that prohibited algebraic manipulation to be a subroutine package was the difficulty of memory management. Fortunately, it is conquered by employing the garbage collection method devised by Boehm and Weiser[3]. It provides `gc_malloc()` function as a storage allocator with automatic garbage collection, which can be used as an alternative of `malloc()`. Furthermore, mixed use of `gc_malloc()` and `malloc()` is allowed provided that a pointer obtained by `gc_malloc()` is not stored in a storage allocated by `malloc()`. Thus, most subroutines calling `malloc()` can coexist with risa libraries.

We mention here another computer algebra system: SAC2-C. It is totally written in C and exists as a UNIX subroutine library. We obtained SAC2-C sources from `archive.cis.ohio-state.edu` by ftp. Though we could not find detailed documentations, it was easy to extract internal expressions from the sources and to write several C functions to convert internal forms between risa and SAC2-C. Then we linked SAC2-C library with asir so that we could call some routines of SAC2-C from risa. The only tests we could do are expansion and factorization of integral polynomials. Although our experience is very limited, its modularity seems to be

sufficiently high and its performance, in part (e.g., univariate factorization), very high.

Many application systems will find benefits when they can rely on an algebraic computation system like risa, PARI and SAC2-C. There is a good example: Our cooperators are now trying to incorporate a part of risa library into their space robot simulator.

2 Structure of risa

2.1 Libraries

Risa, as a set of UNIX subroutine libraries consists of several archive files. Their contents are as follows.

`libca.a` This archive includes functions for fundamental computation such as polynomial arithmetic, GCD, factorization, etc.

`libgc.a` This is the storage manager. It contains a storage allocator and a garbage collector. See Subsec. 1.5 for detail.

`libparse.a` This is the parser. It contains the toplevel parser, the evaluator, the debugger for asir programs and file I/O primitives.

`libtcp.a` This is a collection of functions for interprocess communication.

2.2 Data types

Various data types are defined and used in risa. Among them, there are data types which have object identifiers and are treated as independent objects. Those are as follows. The number of each type corresponds to the object identifier of the type.

1. **number (rational number, floating point number, algebraic number)**
2. **polynomial**
3. **rational function**
4. **list**
5. **vector (one-dimensional array)**
6. **matrix (two-dimensional array)**
7. **string**
8. **structure (currently tested)**

These objects are defined as C structures. As examples, we explain the structures of rational number and polynomial.

A rational number has essentially 4 members: `id`, `sign`, `numerator`, `denominator`. An integer is represented as a rational number whose denominator is equal to 0 (not 1). The type of `numerator` and `denominator` is *natural number*, which is not treated as an independent object, but it is the most fundamental data type in risa. It is represented as an array of machine integers.

A polynomial has essentially 3 members: `id`, `variable`, `exp-coef list`. The member `variable` designates the main variable of the polynomial. The member `exp-coef list` is a list of cells each of which contains an exponent and a coefficient.

Arithmetic operations are defined for each type. Operations between different types are needed not only at the toplevel of the parser but also in the library functions. In risa, a function for an arithmetic operation accepts at its argument place to an object, say `0`, any object whose object identifier is less than or equal to that of `0`. The ordering of object identifiers is determined so that this rule makes sense as much as possible.

2.3 Engine

The engine contains several operations other than arithmetic operations, mainly for polynomials over the rationals. Some of them are as follows.

- polynomial factorization
- square-free factorization
- GCD
- resultant

The algorithm used for univariate factorization is the Berlekamp-Hensel algorithm described in [8]. We did not implement early detection of true factors. Careful implementation of each stage of the algorithm made it work very fast and the univariate factorizer is quite efficient. EZ algorithm with our own improvements is implemented for multivariate factorization and GCD. EEZ algorithm [13] is not implemented yet. For square-free factorization, we use a variant of the algorithm in [12]. This algorithm is especially efficient for a polynomial which has a factor with high multiplicity.

2.4 Storage management

The garbage collector (GC) described in [3] is "conservative", i.e., it does not guarantee that it reclaims, at a garbage collection, all the garbages that the system cannot access. This GC uses static storage, stack and registers as starting points of marking. Then it may happen that an integer accidentally coincides with a valid heap address. This causes a failure of reclaiming some amount of garbages. But if the value is on the stack or on a register, it is expected that the value will have been changed before the next garbage collection and the garbage will eventually be collected. This is practically correct. One disadvantage of this GC method is that it does not make any compaction. It may cause frequent pagings. At present we have no solution for it.

2.5 Parser and evaluator

Asir has two parsers, one for the toplevel and one for the debugger. They are generated from grammar files by `bison` [5] which is compatible with `yacc` [18]. The reason why we use `bison` is that `yacc` allows only one parser in a binary object file. The toplevel parser parses statements and makes a parse tree. The evaluator evaluates the tree and generates an object (internal form).

The evaluator consists of statement evaluator `evalstat()` and expression evaluator `eval()`. Each evaluator interprets identifiers at nodes of the parsed tree and executes corresponding operations. These operations are considered, to some extent, independent of the language. Thus it is fairly easy to change the parser so that it can parse another grammar. Now we are implementing a Maple [4] parser to clarify the parts of the evaluator which depend on asir. We intend to divide the parser library into two separate parts: a language independent part and dependent part. The language dependent part will contain a grammar definition, a lexical analyzer and a minimal part of language dependent subroutines.

2.6 Interprocess communication

The archive `libtcp.a` contains primitives to send and receive internal forms of risa data objects directly between processes. That is, each program linked with this library can continue its own task while passing risa objects to other process. To ensure correct binary data passing independently of the byte order of a machine, **XDR** (external data representation)[17] over TCP/IP stream is used. This mechanism can be used for experiments of simple concurrent/parallel computation. As an example, we implemented a command which executes a process on a machine and opens a stream channel between asir and the process by `libtcp.a`. And as its working example we made a program which draws a graph of an implicit function defined by a bi-variate polynomial. In this example, asir simply acts as a parser and the actual numerical computation is performed on the partner process. Thus if the process is executed on a machine which can carry out numerical computation fast, the execution time is expected to be reduced.

3 Asir

Asir is a program made by linking all of risa libraries and it is a usual interactive computer algebra system. We show a session example.

```
otemoyan: asir
[0] (x+y)^5;
x^5+5*y*x^4+10*y^2*x^3+10*y^3*x^2+5*y^4*x+y^5
[1] for ( I = S = 0; I < 10; I++ ) S += I^2;
[2] S;
285
[3] subst(1/6*n*(2*n+1)*(n+1),n,9);
285
[4] quit;
otemoyan:
```

When the session is interrupted from keyboard, the following menu

```
[10] ^Cinterrupt ?(q/t/c/d)
```

is printed(q:quit,t:toplevel,c:continue,d:debug). If `'d'` is selected, asir automatically enters debug mode after the execution of the current program line. This, for example, is useful for detecting an infinite loop caused by a bug.

3.1 Language

The grammar of asir user language is a variant of that of C. The main differences and the original features as a language for computer algebra are as follows. We call the user language also asir.

- A programming variable begins with an upper case letter and an indeterminate begins with a lower case letter.
- The type declaration of a variable is only needed for **structure**.
- There are two scopes of variables: local variables and global variables. Variables used in a function including its parameters are all local to the function. The variables declared to be external or global are treated as global variables.

- Comma expressions can be used only for **expr**'s in **for (expr;expr;expr)** or **while(expr)**.
- **switch** statement and **goto** statement are not supported.
- Pointer mechanism is not supported.
- A list is expressed as **[expr,expr,...,expr]**.

In order to avoid confusion that an indeterminate is automatically replaced by a value when the indeterminate has the value, asir strictly distinguishes indeterminates from programming variables and assignments to indeterminates are not allowed. In asir, substitution is performed by calling the substitution function.

Except for the above, asir can be regarded as C. Furthermore, asir reads programs through the C preprocessor, and we can use the C preprocessror directives such as `#define` or `#include` in asir programs just like in C programs.

3.2 Debugger

The asir debugger helps to debug asir programs. The debugger is a built-in facility of the parser and the evaluator. In debug mode, asir prints another prompt `(debug)` and accepts commands with its own syntax.

An asir session enters debug mode when

- a debug command `debug` is executed at the toplevel, or
- execution is interrupted from keyboard and then `'d'` option is selected from the interrupt menu.

And the session automatically enters debug mode when

- the evaluator reaches a breakpoint, or
- a step execution command is executed in debug mode, or
- the function `error()` is called in the engine or in the evaluator.

The function `error()` is called when an internal error such as division by zero is detected. And by virtue of the effect of this function, we can examine a value of a programming variable upon an error.

The debugger has a subset of dbx commands. We show some of them.

- `step`

  Execute the next statement. If the statement includes an asir function, `step` steps into the function.

- `next`

  Execute the next statement.

- `print` *exprlist*

  Print the values of elements in *exprlist*. *exprlist* is a comma-separated list of expressions. For each expression in *exprlist* we can use the same syntax as at the toplevel.

- `stop at` *sourceline* [`if` *cond*]

- `stop in` *function* [`if` *cond*]

  Set a breakpoint at *sourceline* or at the top of *function*. If no '`if` *cond*' option follows, then asir enters debug mode whenever execution reaches at the break point; if '`if` *cond*' follows, asir enters debug mode only if the result of evaluation of *cond* is not equal to zero.

- `where`

  Print the calling sequence of functions from toplevel to the current breakpoint.

Other commands include `delete`(remove breakpoints), `status`(show the list of breakpoints), `cont`(continue execution), `list`(list the source program).

When `stop` command is given, the parser searches the program tree for the corresponding node specified by *sourceline* or *function* and inserts a breakpoint statement before the node. Step execution is rather complicated. To recognize the step command, the statement evaluator `evalstat()` checks a step execution flag on each evaluation of a statement. This check corresponds to a test of a memory location at machine-level with a slight overhead, but its affect on the whole execution is very small.

Though it makes debugging quite smooth that we can examine a value of a variable in debug mode, it may happen that we cannot find the data, which is really needed, in the printed value because the printed value itself is often vast. Lists are often conveniently used to represent a structured data. Unfortunately lists are inconvenient when one wish to find the values of its elements by mnemonic names. Currently we are implementing and testing **structure** of C. The **structure** not only makes it easy to read and write programs but also enables us to refer a field of a structured data by its name at debugging.

3.3 Loading object files

Asir supports dynamic loading of C programs. The following C program example shows how to add a new asir function `sqfr()` by calling a C function `sqfrp()` which is in `libca.a`, at the toplevel of asir.

```
#include "ca.h"
#include "parse.h"

int Psqfr();

static reg_sysf() {
    appendubinf("sqfr",Psqfr,1);
}

Psqfr(arg,rp)
NODE arg;
LIST *rp;
{
    DCP dc;
    sqfrp(CO,ARG0(arg),&dc); dcptolist(dc,rp);
}
```

By compiling this into an object file and then loading it into running asir, a command `sqfr` becomes available as an asir function in the asir session.

```
otemoyan: asir
[0] X = (x^2+1)*(x-1)^2*(x+1)^2*(x-3)^3;
x^9-9*x^8+26*x^7-18*x^6-28*x^5+36*x^4-26*x^3
+18*x^2+27*x-27
[1] sqfr(X);
sqfr undefined
0
[2] load("o/sqfr.o")$
[3] sqfr(X);
[[1,1],[x^2+1,1],[x^2-1,2],[x-3,3]]
```

In order to realize this feature, some function of dynamic link editing is needed. So this depends on an operating system. Fortunately, most of `ld`'s (link editor) in BSD UNIX operating systems support dynamic link editing and we can utilize them. See [15] for the details.

4 Porting

At present risa/asir is running on the following CPUs and machines.

- `VAX (microVAX/ultrix)`
- `MC680X0 (X` $\geq$ `2; sun3, NeXT, NEWS, apollo)`
- `SPARC (sun4)`
- `R2000, R3000 (DECStation, RISC NEWS)`
- `MC88100 (LUNA88K)`

Their OS's are all variants of BSD UNIX (Mach for NeXT and LUNA88K). The parts dependent on machines and OS's in risa/asir are as follows.

- double-precision integer multiplication/division
- storage manager
- dynamic object loader

These parts depend on machines and/or OS's because the first two include some assembly language routines and the last two depends on facilities offered by a respective OS. To implement risa/asir on a new machine, these three parts must, in principle, be newly written. If the CPU, however, of the new target machine is one of the above, the existing assembly language routines written for that CPU is often available without modification or at least available as templates. Furthermore, if the OS is BSD UNIX, it is expected that all the existing parts can be used with little modification.

5 Examples

We show some programs and examples of debugging. The first example is an asir program to calculate Catalan's constant. We converted a Maple program in [7] into asir. `idiv()` is a built-in function to get an integer quotient. The operation to convert the result represented by integer into a bigfloat is omitted because asir has no bigfloat routine.

```
def catalan(D) {
    S = T = P = idiv(10^D,2);
    for ( I = J = 1; T; I++ ) {
        J += 2;
        P = idiv(P*I,J);
        T = idiv(T*I+P,J);
        S += T;
    }
    return S;
}
```

The next example is a program to append a list A at the tail of a list B. `car(),cdr(),cons()` are the same functions as those in Lisp. `[]` is a null list.

```
def append(A,B)
{
    if ( A == [] )
        return B;
    else
        return cons(car(A),append(cdr(A),B));
}
```

The next example shows the usage of the debugger. As an example, We use an asir program which calculate factorials recursively.

```
otemoyan: asir                          <-- invoke asir from UNIX shell
[0] load("fac")$                        <-- load an asir program file
factorial() defined.
[3] debug$                              <-- enter debug mode
(debug) list factorial                  <-- list the source program
1   def factorial(X) {
2       if ( !X )
3           return 1;
4       else
5           return X * factorial(X - 1);
6   }
7   end$
(debug) stop at 5                       <-- set a break point
(0) stop at "./fac":5
(debug) quit                            <-- exit debug mode
[4] factorial(6);                       <-- call factorial(6) (should be 6! = 720)
stopped in factorial at line 5 in file "./fac"
5           return X * factorial(X - 1);
(debug) where                           <-- show the current stopping position
factorial(), line 5 in "./fac"
(debug) print X                         <-- print the value of X
X = 6
(debug) step                            <-- execute the next one line (steps into factorial())
stopped in factorial at line 2 in file "./fac"
2 if ( !X )
(debug) where
factorial(), line 2 in "./fac"
factorial(), line 5 in "./fac"
(debug) print X
X = 5
(debug) delete 0                        <-- remove the break point
(debug) cont                            <-- continue execution
720                                     <-- 6!
[5] quit;                               <-- terminate the session
otemoyan:
```

6 Timings

We used SPARCstation 2 (40MHz SPARC, 64Mbytes of main memory) to measure timing data. 16Mbytes of memory was assigned as the initial heap area of asir. The storage manager used in risa determines whether it expands the heap or not by the ratio of the amount of reclaimed storage by a garbage collection and the current heap size. Then, even if the initial heap area is small, the heap is not expanded if sufficient storage is reclaimed by GC. In this case, compared to the case of large initial heap the frequency of garbage collection increases. Thus the total cputime depends on the heap size. So we fixed the initial heap size in each measurement of timing data. It may happen that a particular timing becomes abnormally big due to invocations of garbage collection during the measurement. (Note that cost of garbage collection is rather high.) To average this anomaly, each calculation was repeated several times, and the average execution time, excluding GC time, is shown in a table entry. In the following tables the timings are given in milliseconds.

6.1 Fundamental calculation

6.1.1 Factorials of integers

The following asir functions both calculate factorials of integers.

```
def fact(N) {
/* by repetition */
    for ( I = M = 1; I < N; I++ )
        M *= I;
    return M;
}

def recfact(N) {
/* by recursion */
    return !N ? 1 : N*recfact(N-1);
}
```

In the following table, **fac()** is a built-in function to calculate factorials.

n	1000	2000	5000	10000
fact(n)	551	1891	12841	55420‡
recfact(n)	661	2126	14427	63000‡
fac(n)	199	857	6181	27370‡

GC† : 6.5%

6.1.2 Integer division

The time to calculate factorials is not included.

	2000!/1000!	5000!/2000!	10000!/5000!
time	275	2112	10885

GC† : 0%

6.1.3 Powers of polynomials

n	20	30	40	50
$(a+b+c)^n$	21	47	86	133
$(a+b+c+d)^n$	202	582	1300	2530
$(a+b+c+d+e)^n$	1120	5538	18289	43730‡

GC† : 44.4%

6.2 Factorization

6.2.1 Factorization over the rationals

Here we compare timings of risa with those of REDUCE3.4. It is difficult to compare performances with other systems because it is hard to provide a fair environment. For example, the behavior of REDUCE depends on the lisp on which it is running, and even if a system is written in C, its performance depends on storage availability.

Moreover, as stated in [9], reported timings can show only a *snap-shot* of the factorizer's behavior. Therefore, it is sometimes misleading to compare the timings with those of other systems.

However, in order to show our current achievement, comparison with famous systems is necessary. We used REDUCE3.4 on SPARC obtained from Forbs Systems Inc., Japan. Its Lisp system is an implementation of SLISP, originally developed by A. C. Norman et al. We provide the REDUCE with 16Mbytes of heap area.

Factorization of 19 examples in `factor.tst` in REDUCE3.4.

number	1	2	3	4	5	6
risa§	108	79	102	879	720	1670
REDUCE¶	270	160	200	540	910	1220

7	8	9	10	11	12	13	14
56	7362	3369	1328	3506	12	245	239
310	23030	2270	2010	3020	70	240	760

15	16	17	18	19
238	1902	233	210	324
470	5080	1110	1820	1969

GC(risa)† : 15.5%

Factorization of 10 examples in [14].

number	1	2	3	4	5
risa§	19	155	99	504	2263
REDUCE¶	31	380	470	3940	23170

6	7	8	9	10
2538	100	271	331	950
47360	950	2409	3181	12160

GC(risa)† : 16.1%

6.2.2 Univariate factorization over algebraic number fields

For univariate factorization over algebraic number fields, we use a modification of Trager's method [12]. The algorithm is as follows.

†GC is the ratio of the total GCtime to the whole execution time.
‡indicates that the time was obtained by a single execution, while no mark indicates that the time is the average of 10 executions.
§The time is the average of 10 executions.
¶The time was obtained by a single execution.

$\mathbb{Q}(\alpha) = \mathbb{Q}[t]/(g(t))$ ($g(t) \in \mathbb{Q}[t]$; $g(t)$ is irreducible)

input : square-free $f(x, \alpha) \in \mathbb{Q}(\alpha)[x]$, $s \in \mathbb{Z}$
output : irreducible factors of $f(x)$ over $\mathbb{Q}(\alpha)$

(1) $r(x) = resultant_t(f(x - st, t), g(t))$
(This is the norm of $f(x - s\alpha)$.)
(2) Factorize $r(x)$ over the rationals.
$r(x) = \prod_{i=1}^{m} r_i(x)^{m_i}$
(3) For each i do
(4) Compute $f_i(x) = gcd(f(x), r_i(x + s\alpha))$.
If m_i is equal to 1 then
f_i is an irreducible factor of f
else
apply this algorithm to f_i and $s + 1$.

In the original algorithm $r(x)$ in (1) should be square-free. As described in [12], the norm of an irreducible factor of $f(x)$ is a power of an irreducible polynomial over $\mathbb{Q}$. Then some power of each $r_i(x)$ is a norm of an irreducible factor of $f(x - s\alpha)$. That is, $r_i(x + s\alpha)$ contains an irreducible factor of $f(x)$. In general, it is possible that one $r_i(x + s\alpha)$ contains several factors of $f(x)$, but if the multiplicity of $r_i(x)$ is 1, it contains only one factor. Hence we can use non square-free $r(x)$.

In the worst case, it may happen that $f_i(x)$ in (4) coincides $f(x)$ itself. But it makes no problem because the algorithm is automatically re-executed for the same $f(x)$ and another s.

When $r(x)$ is not square-free, the effects of this modification are as follows.

- We can make full use of the resultant even the case where original algorithm put it into the trash. Note that resultant calculation is very expensive.
- We can apply square-free decomposition before we factorize $r(x)$, so that the total cost of factorization is decreased.
- When we recursively apply this algorithm to a multiple factor, considerable problem size reduction is expected.

Besides, an irreducible factor of a non-multiple factor obtained by square-free decomposition is a norm of some irreducible factor of $f(x - s\alpha)$; then the degree of the irreducible factor is a multiple of the degree of g. So, we can exclude the candidate whose degree is not divisible by the degree of g in the trial division stage of univariate factorization. This observation is also effective for the original algorithm.

There are at most finite numbers of s for which $r(x)$ is not square-free, so it is only a few cases that this modification is effective. But it is useful for a polynomial which yields a non-square-free $r(x)$ for some integer s. Examples listed in [1] are such polynomials.

We factorized some examples in [1] by the original algorithm and the modified algorithm. About 80 % of the time for the factorization of g_5 was spent on the calculation of a resultant. If g_5 is factorized by the original algorithm, we must factorize a polynomial over the integers of degree 256 which has 16 irreducible factors of degree 16 and the modular image of which splits into 128 factors of degree 2. This is a prohibitive number in the practical sense.

Factorization of $g_i(x)$ over $(\mathbb{Q}[t]/(g_i(t)))[x]$ Programs are written in asir language.

number	1	2	3	4	5
modified	250	2150	9300	1740	535460
original	370	6870	325300	3260	–

The time was obtained by a single execution.

$g_1 = x^4 - x + 1$
$g_2 = x^6 + 3x^5 + 6x^4 + x^3 - 3x^2 + 12x + 16$
$g_3 = x^9 - 15x^6 - 87x^3 - 125$
$g_4 = x^9 - 54$
$g_5 = x^{16} - 136x^{14} + 6476x^{12} - 141912x^{10} + 1513334x^8$
$-7453176x^6 + 13950764x^4 - 5596840x^2 + 46225$

7 Conclusion

Risa/asir computer algebra system is intended not only for conventional closed use by interactive end users, but also for more intensive and open use by many scientific application programs. Moreover, it is intended to help users solve problems by cooperating with many other external programs. For this purpose, it is developed on a standard UNIX platform, written in C, and with special emphasis on modularity of its components, efficiency of time and space, and self-containedness. It shows competent, sometimes outstanding, potential for fundamental computation. Many immature points as a conventional computer algebra system shall be improved in the future by risa's affinity to other softwares. Furthermore, we hope that systems which share the same design philosophy as risa constitute an integrated scientific computing system in the future.

References

[1] Abott, J.A., Bradford, R.J.,Davenport, J.H., Factorisation of Polynomials: Old Ideas and Recent Results. Trends in Computer Algebra, LNCS 296 (1988), 81-91.

[2] Batut, C., Bernardi, D., Cohen, H., Olivier, M., User's Guide to PARI-GP. February 1991.

[3] Boehm, H.,Weiser, M., Garbage Collection in an Uncooperative Environment. Software Practice & Experience(September 1988), 807-820.

[4] Char, B.W., Geddes, K.O., Gonnet, G.H., Leong, B.L., Monagan, M.B., Watt, S.M., Maple Language Reference Manual. Walerloo Maple Publishing (1991).

[5] Donnelly, C., Stallman, R., BISON The YACC-compatible Parser Generator. Free Software Foundation (1990).

[6] Fateman, R.J., Advances and Trends in the Design and Construction of Algebraic Systems. Proc. ISSAC '90, 60-67.

[7] Fee, G.J, Computation of Catalan's Constant Using Ramanujan's Formula. Proc. ISSAC '90, 157-160.

[8] Kaltofen, E., Factorization of Polynomials. Computing, Suppl. 4(1982), 95-113.

[9] Moore, P.M.A., Norman, A.C., Implementing a Polynomial Factorization and GCD Package. Proc. SYMSAC 81, 109-116.

[10] SAC2-C source programs. Ftp'ed from `archive.cis.ohio-state.edu`.

[11] Takeshima, T., Noro, M., Yokoyama, K., Report on Computer Algebra Research. FUJITSU Sci. Tech. J., 27,4, December 1991, 338-359.

[12] Trager, B.M., Algebraic Factoring and Rational Function Integration. Proc. SYMSAC 76, 219-226.

[13] Wang, P.S., An Improved Multivariate Polynomial Factoring Algorithm. Math. Comp. 32(1978), 1215-1231.

[14] Wang, P.S., Trager, B.M., New Algorithms for Polynomial Square-Free Decomposition over the Integers. SIAM J. Comp. 8(1979), 300-305.

[15] Yuasa, T., Hagiya, M., The Kyoto Common Lisp Report. Research Institute for Mathematical Sciences, Kyoto University, June 1985.

[16] Debugging Tools. Sun Microsystems, Inc. (1988).

[17] Network Programming. Sun Microsystems, Inc. (1988).

[18] Programming Utilities & Libraries. Sun Microsystems, Inc. (1988).