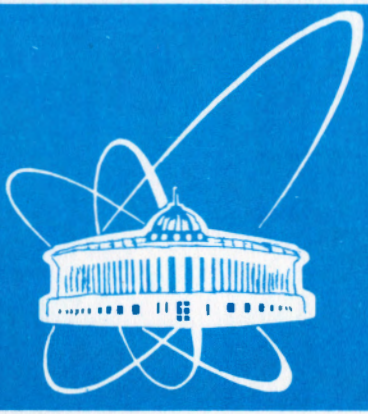

СООБЩЕНИЯ
ОБЪЕДИНЕННОГО
ИНСТИТУТА
ЯДЕРНЫХ
ИССЛЕДОВАНИЙ

Дубна

P11-99-230

А.М.Рапортиренко

ИСПОЛЬЗОВАНИЕ GSL В СИМВОЛЬНЫХ И ЧИСЛЕННЫХ ВЫЧИСЛЕНИЯХ

1999

Рапортиренко А.М.
P11-99-230

Использование GSL в символьных и численных вычислениях

Дано описание реализации численно-символьного интерфейса интерпретатора языка Standard LISP—GSL и средств отладки и анализа временных характеристик динамически загружаемых объектных модулей программ для проведения символьных и численных вычислений. Это дает возможность использовать в символьно-численных вычислениях большое количество программ, написанных на языках C и FORTRAN.

Работа выполнена в Лаборатории вычислительной техники и автоматизации ОИЯИ.

Сообщение Объединенного института ядерных исследований. Дубна, 1999

Raportirenko A.M.
P11-99-230

Use of GSL in Symbolic and Numerical Computations

This paper describes a symbolic-numerical interface of the Standard LISP interpreter—GSL together with debugging and profiling tools of the dynamically loaded object modules for symbolic or numerical computations. This allows one to use a reach number of numerical routines written in C and FORTRAN in symbolic-numerical computations.

The investigation has been performed at the Laboratory of Computing Techniques and Automation, JINR.

Communication of the Joint Institute for Nuclear Research. Dubna, 1999

1. Введение

Диалект языка LISP, Standard LISP [1] был создан в процессе проведения работ над системой аналитических вычислений REDUCE [2]. В нем реализован минимальный набор функций, достаточный для эффективного программирования символьных алгоритмов.

Однако многие задачи компьютерной алгебры приводят к необходимости одновременного проведения как символьных, так и численных вычислений. В принципе, можно программировать на языке LISP и численные алгоритмы. Так как LISP-интерпретаторы обычно используют арифметику неограниченной точности, то это, в свою очередь, позволит избежать многих проблем, связанных с переполнениями и ошибками округления. И это, пожалуй, один из наиболее убедительных доводов в пользу языка LISP. Однако даже если не принимать во внимание тот факт, что и C-, и FORTRAN-программы все-таки должны работать быстрее аналогичных скомпилированных LISP-программ, имеется не менее убедительный довод другого сорта—огромное количество хорошо отлаженных программ численных вычислений, написанных на языках C и FORTRAN.

Попытка сделать доступным пользователю языка LISP программы, написанные на других языках, приводит к необходимости расширения интерпретатора, по крайней мере, для решения следующих вопросов:

- должна быть обеспечена возможность для динамического подключения объектного модуля программы к выполняемому процессу и, по мере возможности, обратная операция. При этом объектный модуль может находиться как в обычной или разделяемой библиотеке, так и просто в виде объектного файла;

- что касается соглашения о связях, то должна быть обеспечена возможность как для вызова программы для проведения численных вычислений из компилированной или интерпретируемой LISP-функции, так и обратная операция—обращение к LISP-функциям (как компилированным, так и интерпретируемым) из программы для проведения численных вычислений;

- в принципе, должна быть обеспечена возможность для отладки и анализа временных характеристик (профилирования) динамически подключенных программ.

Кроме этого, здесь возможны проблемы и чисто программные, например, различный способ обработки особых случаев при операциях с вещественными числами в LISP-интерпретаторе и программе для проведения численных вычислений.

Тем не менее, иметь расширение LISP-интерпретатора для динамического подключения программ проведения численных вычислений весьма полезно. В GSL [5] указанное расширение реализовано наиболее естественным образом, в нем и LISP- и FORTRAN-программы перед компиляцией преобразуются в эквивалентные им C-программы, объектные модули которых обрабатываются одинаково как динамическим загрузчиком, так и другими программами обработки объектных файлов.

2. Интерпретатор

Одной из главных целей проекта GSL [5](GNU Standard LISP) было создание некоммерческого, мобильного, легко расширяемого LISP-интерпретатора, реализующего диалект языка Standard LISP.

Второй, не менее важной целью проекта, было решение вопроса о совместном использовании в одной и той же сессии как динамически загружаемых программ для выполнения численных вычислений, так и динамически загружаемых программ для выполнения символьных вычислений.

Интерпретатор GSL написан на языке C с использованием особенностей компилятора GNU CC [6]. При разработке интерпретатора использовалось только свободное программное обеспечение фирмы Free Software Foundation, разработанное в рамках проекта GNU.

Интерпретатор может быть собран статически или динамически, т.е. с использованием разделяемых системных библиотек. Более того, из самого интерпретатора* можно сформировать статическую или динамическую библиотеку. Как уже упоминалось, GSL имеет компилятор и динамический загрузчик. Динамический загрузчик дает возможность подключать к выполняемому процессу объектные модули, которые могут находиться как в обычных или разделяемых библиотеках, так и просто в виде объектных файлов. Вместе с функцией `dump-gsl`, которая позволяет сбросить образ памяти интерпретатора GSL в исполняемый файл и в дальнейшем начинать работу с него, он

*За исключением модуля, содержащего main-программу.

часто используется для генерации специализированных LISP-систем, которые требуют динамической загрузки большого количества объектных модулей.

В настоящее время интерпретатор работает на различных платформах в BSD-подобных операционных системах UNIX, таких, как NetBSD/(sparc, i386), FreeBSD/i386, SunOS/(sparc, i386), Linux/(sparc, i386). Это связано с тем, что как динамический загрузчик, так и функция `dump-gsl` пока могут работать только с объектными файлами в формате `a.out` и `elf`.

3. Компиляция

Компиляция LISP-функций, т.е. перевод их определяющих выражений в машинные подпрограммы, значительно ускоряет вычисление их значений. Так, сравнение времен выполнения тестовых программ системы REDUCE 3.5, работающей под управлением GSL, показало, что компилированный вариант системы работает в 10–12 раз быстрее, чем в режиме интерпретации.

Процесс компиляци LISP-программ происходит в три этапа. Вначале с помощью Standard LISP-компилятора [3] для каждой функции исходной программы производится генерация последовательности c-макросов (команд абстрактной машины), которые затем используются для генерации C-программы, полностью эквивалентной компилируемой LISP-программе. Полученные таким образом C-программы являются машино-независимыми. Для генерации объектных модулей используется GNU CC [6]-компилятор.

Так как динамический загрузчик игнорирует отладочную информацию, имеющуюся в объектном модуле, то полезно вызвать загрузчик для удаления из объектных файлов отладочной информации и локальных символов. С одной стороны, это позволяет уменьшить размер объектного файла, а с другой, что более существенно, уменьшает количество символов в таблице символов динамического загрузчика.

В качестве примера рассмотрим компиляцию LISP-программы, состоящей из функции вычисления факториала положительного числа:

```
(de factorial (n)
  (cond ((onep n) 1)
    (t (times2 n (factorial (sub1 n))))))
```

Функции `factorial` соответствует следующая последовательность c-макросов:

```
(!*entry factorial expr 1)
(!*alloc 1)
(!*store 1 0)
(!*link onep expr 1)
(!*jumpnil g000003)
(!*load 1 (quote 1))
```

```
(!*jump g000004)
(!*lbl g000003)
(!*load 1 0)
(!*link sub1 expr 1)
(!*link factorial expr 1)
(!*load 2 1)
(!*load 1 0)
(!*link times2 expr 2)
(!*lbl g000004)
(!*dealloc 1)
(!*exit)
```

которые затем используются для генерации C-программы вычисления факториала. Текст сгенерированной C-программы вычисления факториала представлен ниже:

```
/*  Module: factorial */

#include  "GSLtoC.h"

extern Lisp_Object Fsub1 ();

static Lisp_Object _Ffactorial ();
static Lisp_Object Qfactorial;

Lisp_Object *Stat_factorial[1];
struct mframe Objs_factorial;

DE ("factorial", _Ffactorial, Sfactorial, 1,
"       Compiled function.")
  (a1)
     Lisp_Object a1;
{
  register Lisp_Object r1, r2;

  FRAME (1)

  TRACE_ENTRY (Qfactorial, 1, &a1)

   r1 = a1;

  XSETFRM (0, r1);
  r1 = Fonep (r1);
  if (NILP (r1))
    goto g000003;
  r1 = XUINT (1);
```



```
  goto g000004;
g000003:
  r1 = XFRM (0);
  r1 = Fsub1 (r1);
  r1 = _Ffactorial (r1);
  r2 = r1;
  r1 = XFRM (0);
  r1 = Ftimes2 (r1, r2);
g000004:
  UNFRAME (1)

  TRACE_EXIT (Qfactorial, r1);
}

void
Init_factorial ()
{
  Objs_factorial.size = 0;
  Objs_factorial.objs = Stat_factorial;
  Objs_factorial.next = objslist;
  objslist = &Objs_factorial;

  Stat_factorial [0] = &Qnil;
  defentry (&Objs_factorial, &Qfactorial, &Sfactorial, "Ffactorial");
}

/* End of File */
```

4. Динамическая загрузка

Основу динамического загрузчика GSL составляет GNU dld [7, 8]. Для того чтобы сделать возможным доступ не только к объектным модулям статических библиотек, но и к модулям разделяемых библиотек, а также разрешить переопределение LISP-функций* при загрузке объектных файлов компилированных LISP-программ, в dld были внесены некоторые изменения.

При загрузке объектного файла какой-нибудь компилированной LISP-программы, в дополнение к обычным операциям по разрешению неопределенных ссылок, производится вычисление всех S-выражений, аналогичное тому, которое происходит при чтении исходной LISP-программы. Это происходит при вызове функции с именем `Init_<module_name>`. Для того чтобы

*Обычно в случае двойного определения символа выдается сообщение об ошибке.

динамический загрузчик смог отличать объектные файлы LISP-программ от остальных, их имена имеют расширение .g, а не .o, как общепринято.

Ниже представлен пример использования динамического загрузчика. Вначале производится загрузка модуля factorial и вычисляется значение 50!. Далее загружается модуль demo, содержащий программу с именем hello, которая при вызове должна выдать сообщение Hello World, используя обращение к функции printf, находящейся в системной библиотеке libc:

```
/* Module: demo */

#include "GSLtoC.h"
#include "f2c.h"

static Lisp_Object Fferdr ();
static Lisp_Object Fhello ();

static Lisp_Object Qferdr;
static Lisp_Object Qhello;

Lisp_Object *Stat_demo[2];
struct mframe Objs_demo;

double dferdr_ (double *x, long *k);

DE ("ferdr", Fferdr, Sferdr, 2,
"       FERDR(X:floating, K:integer):floating\n\
        Type: EVAL, SPREAD\n\
        Returns value of the Fermi-Dirac function\n\
        for real argument X, and K = -1, 1, 3.")
  (a1, a2)
     Lisp_Object a1, a2;
{
  doublereal x;
  integer k;

  x = l2c_double (a1);
  k = l2c_long (a2);

  return c2l_double(dferdr_ (&x, &k));
}

DE ("hello", Fhello, Shello, 0,
"       Compiled function.")
  ()
{
```



```
  printf ("\n\n\t\tHello World\n");
  return Qnil;
}

void
Init_demo ()
{
  Objs_demo.size = 0;
  Objs_demo.objs = Stat_demo;
  Objs_demo.next = objslist;
  objslist = &Objs_demo;

  defentry (&Objs_demo, &Qferdr, &Sferdr, "Fferdr");
  defentry (&Objs_demo, &Qhello, &Shello, "Fhello");
}

/* End of File */
```

Первоначально символ printf не определен, и если произвести обращение к функции hello, то будет выдано сообщение об ошибке. После разрешения этой неопределенной ссылки с использованием разделяемой библиотеки /usr/lib/libc.so.12.3 обращение к функции hello выдаст ожидаемое сообщение. Обращение к LISP-функции dld_unlink_by_symbol, которая является ни чем иным как GSL-оболочкой для функции с тем же названием, определенной в пакете GNU dld [7], делает символ printf снова не определенным, что видно из сообщения об ошибке при повторном обращении к функции hello:

```
> (load factorial)
t
> (factorial 50)
30414093201713378043612608166064768844377641568960512000000000000
> (load demo)
t
> (list-undefined)
There are a total of 5 undefined symbols:
        _dferdr_
        _l2c_long
        _c2l_double
        _l2c_double
        _printf
nil
> (dld_link "/usr/lib/libc.so.12.3")
t
> (list-undefined)
```

```
There are a total of 4 undefined symbols:
        _dferdr_
        _l2c_long
        _c2l_double
        _l2c_double
nil
> (hello)


                Hello World
nil
> (dld_unlink_by_symbol "printf" t)
t
> (hello)
 ***** undefined symbol
(hello)
> (list-undefined)
There are a total of 5 undefined symbols:
        _dferdr_
        _l2c_long
        _c2l_double
        _l2c_double
        _printf
nil
```

5. Численно-символьный интерфейс

Проблемы, которые могут возникнуть при использовании в GSL программ для проведения численных вычислений, чаще всего связаны с неправильным преобразованием аргументов и/или возвращаемого значения при переходах между LISP- и C- или FORTRAN-программами. Это связано с тем, что в GSL используется внутреннее представление целых чисел, отличное от стандартного.

В зависимости от значения целое число может иметь одно из следующих представлений:

- если абсолютная величина числа меньше чем $2^{\texttt{addr\_size}-1}$, то для его представления используется сам указатель: значение числа помещается в поле адреса указателя. Ширина поля адреса `addr_size` не фиксирована и может быть изменена перед компиляцией интерпретатора. Для представления значения числа используется дополнительный код;
- если абсолютная величина числа больше или равна $2^{\texttt{addr\_size}-1}$ и меньше чем $2^{\texttt{sizeof(longlong)}-2}$, то для его представления используются два смежных машинных слова. Значение числа также представлено в дополнительном коде;
- для представления целого числа, абсолютная величина которого не меньше чем $2^{\texttt{sizeof(longlong)}-2}$, используется разложение по основанию $b = 2^{\texttt{sizeof(long)}}$:

$$c_0 + c_1 b^1 + ... + c_n b^n, c_i < b$$

Значение большого целого представлено в прямом коде в виде массива коэффициентов разложения c_i. Для работы с большими целыми в GSL используется пакет GNU gmp [9], в котором реализована высокоэффективная арифметика многократной точности.

Что касается вещественных чисел, то они соответствуют типу double языка C и обычно имеют представление, соответствующее IEEE-формату для двойной точности.

Следует также отметить, что в GSL как целый, так и вещественный нули представлены одним и тем же числом—коротким целым со значением ноль.

С другой стороны, программа преобразования FORTRAN в C f2c [4] использует соответствие между типами, показанное в таблице. Для того чтобы

FORTRAN	C	standard f2c.h
`integer*2 x`	`shortint x;`	`short int x;`
`integer x`	`integer x;`	`long int x;`
`logical x`	`long int x;`	`long int x;`
`real x`	`real x;`	`float x;`
`double precision x`	`doublereal x;`	`double x;`
`character*6 x`	`char x[6];`	`char x[6];`
`complex x`	`complex x;`	`struct {float r, i;} x;`
`double complex x`	`doublecomplex x;`	`struct {double r, i;} x;`

правильно преобразовывать аргументы при совместной работе программ для проведения как символьных, так и численных вычислений, необходимо использовать функции, прототипы которых представлены ниже:

```
short int   l2c_short (Lisp_Object num);
Lisp_Object c2l_short (short int n);

long int    l2c_long (Lisp_Object num);
Lisp_Object c2l_long (long int n);

double      l2c_double (Lisp_Object num);
Lisp_Object c2l_double (double n);
```

```
char *      l2c_string (Lisp_Object str);
Lisp_Object c2l_string build_string (char *str);
```

Пример использования некоторых из них можно увидеть в программе `ferdr` модуля `demo`.

Тип `complex` в GSL не определен, и потому, как выбор его внутреннего представления, так и программирование основных арифметических операций над комплексными числами зависит от пользователя.

6. Отладка динамически загружаемых программ

Так как средства трассировки LISP-интерпретатора могут быть использованы только для отладки как компилированных, так и интерпретируемых LISP-программ, то возникал очень непростой вопрос: как быть с отладкой динамически загружаемых программ для проведения численных вычислений? К счастью, GNU gdb [10] позволяет динамически подгружать таблицы символов объектных файлов и потому вполне может быть использован для отладки динамически загружаемых программ. Ниже, на примере использования в GSL FORTRAN-программы для вычисления функции Ферми-Дирака*

$$F_k(x) = \int_0^\infty \frac{t^{k/2}}{1+e^{t-x}}dt$$

для вещественного аргумента x, и $k = -1, 1, 3$, проводится демонстрация использования gdb для отладки динамически загруженной программы `demo`. Как уже упоминалось ранее, динамический загрузчик игнорирует отладочную информацию, имеющуюся в объектном модуле. Поэтому при формировании `.g`-файла мы вызываем загрузчик для удаления из объектного файла отладочной информации и локальных символов. Модуль `GSLf2c` содержит функции преобразования аргументов между LISP- и C-программами. В модуле `dld_tools` определена LISP-функция `list-modules`, которая высвечивает на экране имена динамически загруженных объектных модулей и их начальные адреса в памяти. После определения начального адреса модуля `demo`, с помощью gdb-команды `add-symbol-file` загружается отладочная информация из `.o`-файла, после чего можно заниматься его отладкой:

```
bash$ gcc -c -g -O2 demo.c
bash$ ld -r -x -S -o demo.g demo.o
bash$ gdb /home/GSL-1.0/GSLbin/gsla4
(gdb) run
```

*Исходные тексты этой функции взяты из библиотеки CERNLIB


```
Starting program: /home/GSL-1.0/GSLbin/gsla4

> (load demo GSLf2c dld_tools)
t
> (dld_link "/home/GSL-1.0/GSLlib/libCERN.a")
t
> (dld_link "/usr/gnu/lib/libf2c.a")
t
> (dld_link "/usr/lib/libc.so.12.3")
t
> (dld_link "/usr/lib/libm.a")
t
> (info ferdr)
        FERDR(X:floating, K:integer):floating
        Type: EVAL, SPREAD
        Returns value of the Fermi-Dirac function
        for real argument X, and K = -1, 1, 3.
nil
> (ferdr 1.5 -1)
2.21436797536334
> (ferdr 1.5 -2)
      ***** CERN C323 FRERDR/DFERDR ERROR C323.1: INCORRECT K = -2
0
> (list-modules)
====== library name: /usr/lib/libm.a
        start: 0x79900 name: w_exp.o
        .
        .
        start: 0xc1bc0 name: s_copysign.o
start: 0x0 name: /usr/lib/libc.so.12.3
====== library name: /usr/gnu/lib/libf2c.a
        start: 0x79f00 name: s_cmp.o
        .
        .
        start: 0x8d000 name: err.o
====== library name: /home/GSL-1.0/GSLlib/libCERN.a
        start: 0x7a000 name: ferfr64.o
        start: 0x76600 name: mtlprt.o
        start: 0x7b000 name: mtlset.o
        start: 0x4ca00 name: gsl_abend.o
        start: 0x79e00 name: lenocc.o
start: 0x78000 name: /home/GSL-1.0/GSLlib/dld_tools.g
start: 0x77000 name: /home/GSL-1.0/GSLlib/GSLf2c.g
start: 0x76000 name: /home/ram/xmp/demo.g
start: 0x0 name: -*- dummy entry -*-
```

```
start: 0x0 name: /home/GSL-1.0/GSLbin/gsla4
nil
> ^C
Program received signal SIGINT, Interrupt.
0x40a8ff0 in read ()
(gdb) add-symbol-file /home/ram/xmp/demo.o 0x76000
(gdb) l demo.c:27
Source file is more recent than executable.
22        (a1, a2)
23           Lisp_Object a1, a2;
24      {
25        doublereal x;
26        integer k;
27
28        x = l2c_double (a1);
29        k = l2c_long (a2);
30
31        return c2l_double(dferdr_ (&x, &k));
(gdb) b 30
Breakpoint 1 at 0x760b4: file ./c/demo.c, line 30.
(gdb) c
Continuing.
(ferdr 1.5 -1)

Breakpoint 1, Fferdr (a1=201691152, a2=67108863) at ./c/demo.c:31
31        return c2l_double(dferdr_ (&x, &k));
(gdb) p x
$1 = 1.5
(gdb) p k
$2 = -1
(gdb) q
The program is running.  Quit anyway (and kill it)? (y or n) y
bash$
```

7. Анализ временных характеристик (профилирование) динамически загружаемых программ

Нет необходимости говорить о значении утилиты gprof [11, 12] для анализа временных характеристик различных компонент оптимизируемой программы. Стандартный вариант этой утилиты предполагает, что исследуемый `a.out`-файл содержит информацию о всех символах программы и потому не подходит для работы с динамически загружаемыми программами. В GSL используется модифицированный вариант gprof, в который была добавлена возможность динамически подгружать таблицы символов объектных файлов.

Ниже, на примере использования программы для вычисления факториала положительного числа, проводится демонстрация использования gprof для профилирования динамически загруженной программы `factorial` и функций интерпретатора.
Для того чтобы информация об исследуемом модуле заносилась в файл `gmon.out`, он должен быть скомпилирован с флагом `-pg`:

```
bash$ gcc -c -pg -O2 factorial.c
bash$ ld -r -x -S -o factorial.g factorial.o
```

Затем создается исполняемый файл, который, кроме исследуемых, должен содержать два вспомогательных модуля–`dld_prof` и `dld_tools`:

```
bash$ /home/GSL-1.0/GSLbin/gsla4_p
> (load dld_prof dld_tools factorial)
t
> (dld_link "/usr/lib/libc.so.12.3")
t
> (dump-gsl "gslf" "/home/GSL-1.0/GSLbin/gsla4_p" nil)
nil
> (quit)
bash$
```

После этого производится запуск программы для сбора профилирующей информации, которая будет сбрасываться в файл `gmon.out`. Управление сбором информации производится функцией `dld_moncontrol`:

```
bash$ ./gslf
> (list-modules)
start: 0x0 name: /usr/lib/libc.so.12.3
start: 0x79e00 name: /home/ram/xmp/factorial.g
start: 0x7a000 name: /home/GSL-1.0/GSLlib/dld_tools.g
start: 0x78000 name: /home/GSL-1.0/GSLlib/dld_prof.g
start: 0x0 name: -*- dummy entry -*-
start: 0x0 name: /home/GSL-1.0/GSLbin/gsla4_p
nil
> (dld_monstartup)
nil
> (dld_moncontrol t)
nil
> (factorial 1000)
        .
        .
> (dld_moncontrol nil)
nil
> (dld_moncleanup)
nil
```

```
> (quit)
bash$
```

Далее, для обработки информации, накопленной а файле gmon.out, используется модифицированная версия программы gprof–gslprof:

```
bash$ gslprof -S gslf.gprof ./gslf >& gmon.out.gslf
```

Каждая строка файла gslf.gprof содержит имя файла с таблицей символов и адрес начала соответствующего модуля. В данном случае он состоит из одной строки:

```
add-symbol-file /home/ram/xmp/factorial.o          0x79e00
```

Первое, что бросается в глаза при просмотре фрагментов выдачи программы gprof–было 22 сборки мусора! В этом нет ничего особенного. Дело в том, что в GSL сборщик мусора вызывается всякий раз, когда у системы было получено определенное количество памяти. Начальное значение максимального количества памяти, получаемого между двумя сборками мусора, устанавливается при компиляции интерпретатора и в данном случае равно 20480 байт. Новое значение может быть установлено при вызове функции gc-cons-threshold.

Из 1000 умножений 979 раз короткое целое умножалось на большое целое.

```
granularity: each sample hit covers 4 byte(s) for 1.72% of 0.58 seconds

  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 37.9      0.22     0.22        1   220.00   220.00  _mpz2str [9]
 25.9      0.37     0.15      979     0.15     0.16  _times2_mpz_int [11]
 15.5      0.46     0.09                             _moncontrol (500)
  5.2      0.49     0.03    18902     0.00     0.00  _mark_object [13]
  3.4      0.51     0.02       22     0.91     3.18  _Freclaim [12]
  3.4      0.53     0.02        1    20.00   250.00  __Ffactorial [5]


granularity: each sample hit covers 4 byte(s) for 2.04% of 0.49 seconds

                                 called/total       parents
index %time    self  descendents  called+self    name           index
                                  called/total       children
-----------------------------------------------
                                      999             __Ffactorial [5]
                0.02        0.23       1/1           _Feval <cycle 2> [3]
[5]     51.0    0.02        0.23       1+999     __Ffactorial [5]
```



```
                0.00        0.23     999/999         _gcFtimes2 [6]
                0.00        0.00    1000/1000        _Fonep [28]
                0.00        0.00     999/999         _gcFsub1 [30]
                                        999          __Ffactorial [5]
-----------------------------------------------
                0.00        0.23     999/999         __Ffactorial [5]
[6]     46.9    0.00        0.23       999       _gcFtimes2 [6]
                0.00        0.16     999/999         _Ftimes2 [10]
                0.02        0.05      22/22          _Freclaim [12]
-----------------------------------------------
                0.00        0.16     999/999         _gcFtimes2 [6]
[10]    32.7    0.00        0.16       999       _Ftimes2 [10]
                0.15        0.01     979/979         _times2_mpz_int [11]
                0.00        0.00      10/10          _times2_fix_int [25]
                0.00        0.00       1/10          _make_fix [35]
```

8. Заключение

Представленные в статье средства дают возможность пользователю системы REDUCE [2] с одной стороны использовать в символьных вычислениях большого количества программ для проведения численных вычислений, написанных на языках C или FORTRAN. А с другой, что не менее важно, производить отладку и анализ временных характеристик динамически загружаемых модулей независимо от того предназначены ли они для проведения символьных или численных вычислений. Работа выполнена при частичной поддержке INTAS, грант № INTAS-96-0842.

ЛИТЕРАТУРА

1. J. B. Marti, A. C. Hearn, M. L. Griss, C. Griss,
The Standard Lisp Report, SIGPLAN Notices, ACM, 14, (10), 48-68, (1979).

2. A. C. Hearn,
REDUCE User's Manual, Version 3.6, RAND, Santa Monica, 1995.

3. M. L. Griss, A. C. Hearn,
A Portable LISP Compiler, Software–Practice and experience, 11, (6), 541-605, (1981).

4. S. I. Feldman, David M. Gay, Mark W. Maimone, N. L. Schryer,
A Fortran-to-C Converter, AT&T Bell Laboratories, Computing Science Technical Report No. 149, 1995.

5. A. M. Raportirenko,
GSL: A portable Standard LISP interpreter, In Proceedings of the Third International Workshop on Software Engineering, Artificial Intelligence and Expert Systems for High Energy and Nuclear Physics, 1993.

6. R. M. Stallman,
Using and Porting GNU CC, Free Software Foundation, Cambridge, 1998.

7. W. Wilson Ho,
A Dynamic Link/Unlink Editor, Free Software Foundation, Cambridge, 1991.

8. W. Wilson Ho, R. A. Olsson,
An Approach to Genuine Dynamic Linking, Software–Practice and experience, 21, (4), 375-390, (1991).

9. T. Granlund,
The GNU Multiple Precision Arithmetic Library, Free Software Foundation, Cambridge, 1993.

10. R. M. Stallman,, R. H. Pesch,
The GNU Source-Level Debuger, Free Software Foundation, Cambridge, 1998.

11. J. Fenlason, R. M. Stallman,
The GNU profiler, Free Software Foundation, Cambridge, 1993.

12. S. Graham, P. Kessler, M. McKusick,
gprof: A Call Graph Execution Profiler, SIGPLAN Notices, 6, (17), 120-126, (1982).

Рукопись поступила в издательский отдел
31 августа 1999 года.