УДК 510.676, 519.7

Common Lisp версия системы компьютерной алгебры REDUCE

А. М. Рапортиренко

Лаборатория информационных технологий
Объединённый институт ядерных исследований
ул. Жолио-Кюри д.6, Дубна, Московская область, 141980, Россия

В работе представлена реализация системы компьютерной алгебры *REDUCE* в среде Common Lisp с использованием интерпретаторов *CMUCL* и *GCL*. Рассмотрены вопросы совместного использования программ, написанных на языках Common LISP и Snandard LISP, а также специфичные для *CMUCL* и *GCL* особенности компиляции и оптимизации системы.

Ключевые слова: REDUCE, Common Lisp, компьютерная алгебра.

1. Введение

В последнее время список свободно распространяемых систем компьютерной алгебры пополнился системой *REDUCE* [1]. Теперь она относится к тому же классу свободно распространяемых универсальных систем символьно-численных вычислений, что и *AXIOM* [2] или *MAXIMA* [3].

Однако в отличие от систем компьютерной алгебры *AXIOM* и *MAXIMA*, основой которых является Common LISP, в *REDUCE* используется диалект языка LISP, называемый Standard LISP. Имеются две реализации этого диалекта: Portable Standard LISP [4] (*PSL*) и Codemist Standard LISP [5] (*CSL*) и, соответственно, две различных версии системы *REDUCE*.

Для пользователей системы, работающих исключительно в алгебраической моде, этого вполне достаточно. Стоит только отметить, что *PSL* версия является наиболее оптимальной для работы.

Если же смотреть на *REDUCE* как на библиотеку LISP модулей, реализующих большое количество символьных алгоритмов, то возможности разработчика при программировании на языке LISP существенно расширяются. Поскольку в настоящее время Common LISP является стандартом и все, что разрабатывается, пишется на нем, то представляется вполне разумным сделать *REDUCE* работающим на Common LISP интерпретаторах.

В этом случае, с одной стороны, станет возможным совместное использование функций или модулей, написанных на разных диалектах языка LISP, а с другой, появится возможность обращения к программам, написанным на языке *C*, например для проведения численных расчётов, необходимых в процессе решения своей символьной задачи.

2. Немного о реализации

Чтобы иметь наглядное представление о реализации, ниже приведён небольшой фрагмент *REDUCE* сессии, выполненной на интерпретаторе *CMUCL*:

```
9: lisp standardlisp();
$ cl:*package*
#<The STANDARD-LISP-USER package, 3761/6314 internal, 0/9 external>
$ (de foo (u)
      (progn (terpri)
             (print "standard-lisp-user foo is called")
             (cl-user::foo u)))
FOO
```

Исследования, представленные в работе, были выполнены при поддержке гранта РФФИ 07-01-00660.

```
$ (commonlisp)
% (defun foo (u)
         (print "common-lisp-user foo is called")
         (prin1 u)
         (if (/= u 0) (sl-user::foo (1- u)) "ok"))
FOO
% (sl-user::foo 1)
"standard-lisp-user foo is called"
"common-lisp-user foo is called" 1
"standard-lisp-user foo is called"
"common-lisp-user foo is called" 0
"ok"
% (car 1)
Type-error in KERNEL::OBJECT-NOT-LIST-ERROR-HANDLER:  1 is not of type LIST
   [Condition of type TYPE-ERROR]
Restarts:
  0: [ABORT] Return to Common LISP Top-Level.
  1:         Return to Standard LISP Top-Level.
  2:         Return to REDUCE begin loop.
Debug  (type H for help)
(CAR 1 1)[:EXTERNAL]
Source:
; File: target:code/list.lisp
(DEFUN CAR (LIST) "Returns the 1st object in a list." (CAR LIST))
0] 0
% sl::exit-lisp-loop
```

Вначале, находясь в алгебраической моде системы *REDUCE*, командой standard-lisp запускается Top-Level-Loop интерпретатора Standard LISP. На это указывает промпт `$`, при этом мы автоматически попадаем в пространство имён sl-user. Здесь определяется функция sl-user::foo, которая вызывает функцию cl-user::foo из пакета cl-user.

Затем командой commonlisp запускается Top-Level-Loop интерпретатора Common LISP, изменившийся на `%` промпт указывает на это. Здесь определяется функция cl-user::foo, вызывающая ранее определённую функцию sl-user::foo.

Обе функции имеют одно и то же имя, но определены в разных пакетах. Более того, при вызове они обрабатываются разными интерпретаторами. Приблизительно по такой же схеме может быть организована совместная работа и более сложных конструкций.

Обращение к функции car с неправильным аргументом сигнализирует об ошибке. Система даёт возможность или произвести дальнейший анализ и отладку, или же вернуться в одно из перечисленных состояний.

Ввод имени sl::exit-lisp-loop завершает текущую LISP сессию и возвращает контроль предыдущему уровню. Это может быть алгебраическая мода системы *REDUCE* или один из ранее запущенных Top-Level-Loop интерпретатора.

Из приведённого выше примера видна структура системы. То, что относится к интерпретатору Common LISP, изначально определено в пакете commonlisp (сокращённое имя cl). Все, что определяется для обработки интерпретатором Common LISP, по умолчанию помещается в пакет common-lisp-user (сокращённое имя cl-user). Аналогично, то, что относится к интерпретатору Standard LISP, изначально определено в пакете standard-lisp (сокращённое имя sl). Все, что определяется для обработки интерпретатором Standard LISP, по умолчанию помещается в пакет standard-lisp-user (сокращённое имя sl-user).

Интерпретатор Standard LISP составляют около 350 Common LISP функций. Это полноценный интерпретатор, работающий в среде Common LISP. Для разработки использовались интерпретаторы *CMUCL* [6] и *GCL* [7], работающие на платформе x86 под управлением gentoo linux.

Система *REDUCE* — это примерно 14000 Standard LISP функций. В процессе её адаптации было выявлено 7 синтаксических ошибок в исходных *RLISP* текстах.

При компиляции системы никакой существенной оптимизации компилируемых кодов не производилось. Здесь существует масса возможностей, однако, все упирается в точность измерения времени обращения к профилируемым функциям. Единственное, что было сделано, так это преобразование вызовов к функциям

difference, plus2, equal с двумя аргументами, значением одного из которых было 1 на вызовы соответствующих функций с одним аргументом sub1, add1, onep, а также преобразование вызовов к функциям типа cXXXXr к цепочке обращений к функциям car и cdr. Хотя последнее предполагает замену вызовов подстановкой соответствующих инструкций процессора, это пока сделано не было.

Время выполнения стандартных тестов скомпилированными версиями системы *REDUCE* для обоих интерпретаторов приблизительно одинаково и в 4–4.5 больше, чем у *PSL* версии. Времена, выдаваемые во всех трёх случаях, сильно меняются в зависимости от загрузки системы, но их отношение сохраняется.

Следует отметить, что уровень оптимизации, используемый в *PSL*, приводит к различному поведению интерпретируемых и скомпилированных функций, о чем можно судить из нижеприведённого фрагмента:

```
1: lisp standardlisp();
REDUCE, 17-Mar-2007
1 lisp> (de caar1 (u) (caar u))
caar1
2 lisp> (setq *comp t)
t
3 lisp> (de caar2 (u) (caar u))
Request to set constant bitsperword to a different value.
*** (caar2): base 16#8298C64, length 10#17 bytes
caar2
4 lisp> (getd 'caar1)
(expr lambda (u) (caar u))
5 lisp> (getd 'caar2)
(expr . #<Code 1 298C68>)
6 lisp> (caar1 nil)
***** An attempt was made to do caar on `nil', which is not a pair
7 lisp> (caar2 nil)
***** Segmentation Violation in caar2
```

Это связано с тем, что в случае caar1 идёт обращение к функции интерпретатора caar, длиною не менее 14 инструкций процессора, которой и производится анализ типа аргумента. В то время как caar2 определяется 3 инструкциями, здесь контроля за типом данных уже нет.

Тогда как *CMUCL* версия в обоих случаях отрабатывает ошибки одинаковым образом:

```
1: lisp standardlisp();
$ (de caar1 (u) (caar u))
CAAR1
$ (setq *comp t)
T
$ (de caar2 (u) (caar u))
; Compiling LAMBDA (U):
; Compiling Top-Level Form:
CAAR2
$ (getd 'caar1)
(EXPR . (LAMBDA (U) (CAAR U)))
$ (getd 'caar2)
(EXPR . #<Function CAAR2 {5823CB59}>)
$ (caar1 nil)
***** "NIL is not a pair"
$ (caar2 nil)
***** "NIL is not a pair"
```

Так в общем должно и быть, так как в обоих случаях идёт обращение к функции интерпретатора caar. Более того, в *CMUCL* возможна полноценная отладка как интерпретируемых, так и скомпилированных функций.

Второй причиной потери скорости, скорее всего, является арифметика. К примеру Common LISP версии системы *REDUCE* используют определения элементарных функций из *RLISP* пакета `math.red`, в то время как *PSL* версия похоже их не использует, поскольку результаты разные. Функции, определённые в `math.red`, существенно медленнее, но работают правильно, о чем можно судить, сравнивая результаты вычислений программой diff:

```
--- ../psl/arith.rlg    2008-04-24 15:32:12.000000000 +0400
+++ ../cmu/arith.rlg    2008-04-24 15:17:10.000000000 +0400
```

```
@@ -88,13 +102,13 @@
 z := sin(pi);
-z := 1.2246063538e-16
+Z := 0
@@ -502,7 +516,7 @@
 asin 2;
 1.5707963268 - 1.3169578969*I
 sin ws;
-2.0 - 1.060540212e-16*i
+2.0
@@ -544,7 +558,7 @@
 e**(i*pi);
- - 1 + 5.6654984523e-16*i
+ - 1
@@ -1479,7 +1493,7 @@
 e**(pi*i);
- - 1 + 5.6655e-16*i
+ - 1
@@ -1587,7 +1601,7 @@
 acsc 0.1;
 1.5708 - 2.99322*i
 csc ws;
-0.1 + 6.09234e-18*i
+0.1
@@ -1623,7 +1637,7 @@
 atanh 2;
 0.549306 + 1.5708*i
 tanh ws;
-2.0 + 1.83691e-16*i
+2.0
@@ -1635,7 +1649,7 @@
acoth 0.3;
0.30952 + 1.5708*i
 coth ws;
-0.3 - 5.57196e-17*i
+0.3
--- ../psl/taylor.rlg   2008-04-24 15:34:27.000000000 +0400
+++ ../cmu/taylor.rlg   2008-04-24 15:28:42.000000000 +0400
@@ -1360,11 +1375,8 @@
 taylor(sin x,x,pi/2,4);
-                                                          2
-1 + 6.1230317691e-17*(x - 1.5707963268) - 0.5*(x - 1.5707963268)
-
-                                      3                                   4
- - 1.0205052949e-17*(x - 1.5707963268)  + 0.041666666667*(x - 1.5707963268)
+                          2                 4
+1 - 0.5*(X - 1.5707963268)  + 0.041666666667*(X - 1.5707963268)
```

Как видно, хоть ошибки и незначительные, но они приводят к несуществующим слагаемым в разложении.

3. Заключение

В заключение хотелось бы отметить, что каждая из приведённых систем обладает как своими достоинствами, так и недостатками.

На 32 битовых ПК *PSL* версия системы может использовать не более 128M оперативной памяти. Система, построенная на базе *CMUCL*, позволяет использовать только 512M оперативной памяти, в то время как *GCL* версия может забрать всю, которую ей даст операционная система.

Компилятор *CMUCL* при работе существенно использует описание типов аргументов и результатов функций, в то время как для компилятора *GCL* такая информация не имеет практического значения. В первом случае возникают большие неудобства при компиляции автоматически сгенерённых LISP функций, зато, в принципе, можно получить оптимальный код.

В любом случае, на данный момент обе системы хороши для отладки. Всем желающим автор готов предоставить исходные тексты.

Литература

1. REDUCE. — http://reduce-algebra.com.
2. AXIOM. — http://portal.axiom-developer.org.
3. MAXIMA. — http://maxima.sourceforge.net.
4. PSL. — http://www.zib.de/Symbolik/reduce.
5. CSL. — http://www.codemist.co.uk.
6. CMUCL. — http://www.cons.org/cmucl.
7. GCL. — http://savannah.gnu.org/projects/gcl.

UDC 510.676, 519.7

Common Lisp Version of the System of Computer Algebra REDUCE

A. M. Raportirenko

Laboratory of Information Technologies
Joint Institute for Nuclear Research
Joliot-Curie 6, 141980 Dubna, Moscow region, Russia

The work deals with implementation of the system of computer algebra *REDUCE* in the Common Lisp environment with the use of interpreters *CMUCL* and *GCL*. The issues of sharing the programs written in the programming languages Common LISP and Standard LISP as well as specific for *CMUCL* and *GCL* features of compilation and optimization of the system are considered.

Key words and phrases: REDUCE, Common Lisp, computer algebra.